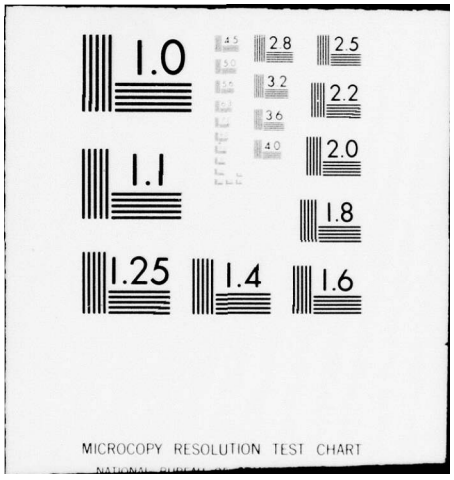
AD-A061 633 IBM THOMAS J WATSON RESEARCH CENTER YORKTOWN HEIGHTS N Y F/G 9/2
BEHAVIORAL STUDIES OF THE PROGRAMMING PROCESS.(U)
OCT 78 L A MILLER N00014-72-C-0419
UNCLASSIFIED NL

1 OF 1
AD A061 633

END
DATE FILMED
2-79
DDC


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
MICROCOPY RESOLUTION TEST CHART

LEVEL II

7

BEHAVIORAL STUDIES OF THE PROGRAMMING PROCESS

Lance A. Miller

IBM Thomas J. Watson Research Center
P. O. Box 218, Yorktown Heights, New York 10598

1p October 1978

78p

Final Report for Period 1 May 1972 - 28 February 1977

Contract N00014-72-C-0419

Sponsored by Office of Naval Research

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Behavioral Studies of the Programming Process		5. TYPE OF REPORT & PERIOD COVERED Period 5/1/72 - 2/28/77 Final Report 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Lance A. Miller		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS I. B. M. Thomas J. Watson Research Center P. O. Box 218, Yorktown Heights, N. Y. 10598		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS Work Unit NR197-020 Code 455
11. CONTROLLING OFFICE NAME AND ADDRESS Engineering Psychology Programs Office of Naval Research		12. REPORT DATE 10-16-78 13. NUMBER OF PAGES 72
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) Unclassified 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Reproduction in whole or in part is permitted for any purpose of the United States Government

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Programming
Software
Design
Natural Language
Routine tasks
Problem-solving
Interactive systems

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This is a final report on an Office of Naval Research contract to investigate the behavioral problems and processes of computer programming. Following an overview of the effort given in section (1), the research work is discussed in sections (2)-(7) under six headings, with Motivation, Work Summary, and Recommendations given for each. Section headings are: (2) computer usage statistics; (3) performance with programming control structures; (4) natural language programming

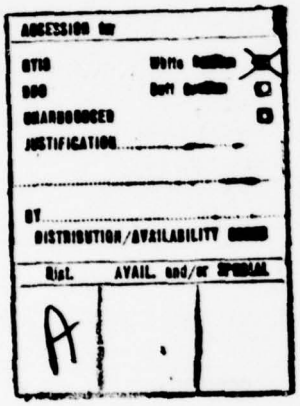
DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

20. Cont.

and communication; (5) investigation of program design behavior; (6) investigation of program modification behavior; and (7) general behavioral issues in interactive computer systems. The published reports, under the contract, are listed in section (8) with the contract personnel given in section (9).



TO: John J. O'Hare, Ph.D., Assistant Director
Engineering Psychology Programs Code 455
Office of Naval Research,
Department of the Navy
Arlington, Virginia 22217

FROM: Lance A. Miller, Ph.D., Manager
Behavioral Sciences Group
Computer Sciences Department
I.B.M. Thomas J. Watson Research Center
Yorktown Heights, New York 10598

SUBJECT: Final Report, "Behavioral Aspects of Computer Programming"

I. This constitutes a final report of work completed under support of Contract N00014-72-C-0419, Work Unit NR197-020, between I.B.M. and Engineering Psychology Programs, Code 455, Office of Naval Research. The principal investigator was Dr. L. A. Miller.

II. The objective of the contract was to conduct research directed at obtaining a basic understanding of the cognitive processes and problems of programming, and thereby to afford the basis for development of tools and methodologies for improving program quality and programmer productivity.

III. The contract work yielded 21 reports (cf. Section 8). Seventeen persons were partially supported at various times, and in various positions, under the contract (cf. Section 9).

IV. The research work is summarized under six major headings, which correspond roughly to successive research foci:
(1). Computer usage statistics.
(2). Performance with programming control structures.
(3). Natural language programming and communication
(4). Investigation of program design behavior.
(5). Investigation of program modification behavior.
(6). General behavioral issues with interactive systems.

V. The discussion of each area covers: (1) the research motivation, (2) detailed summary of the research work, findings, and theory, and (3) our recommendations for specific tools, practices, and further research. The relevant references in Section 8 are indicated in the text by numbers in parentheses.

TABLE OF CONTENTS

1. GENERAL OVERVIEW

In our original contract proposal we suggested "a very general stage model of the start-to-finish aspects of a programming task, as related to more or less established psychological areas of investigation." The stages were: (1) loose initial statement of the problem, (2) precise problem formulation, (3) design of a general solution algorithm, usually in natural language, (4) translation of this solution into a computer program, and (5) debugging, testing, and verification of the program. To these five stages we here add a sixth, which occurs *after* the first five steps have been completed and the program has been accepted by the original requestor, i.e., (6) program maintenance and modification.

Four of our areas of research under the contract are closely related to the stages of this model, with two areas focussing on broader considerations of the programming environment. The six areas of research are presented here roughly in the same order that we performed the work. Thus, in Section 2 we describe the prefacing of our controlled experimental studies with quantitative observational analyses of user performance on interactive systems. In Section 3 we present our initial following of the stage-model, first concentrating on the stage-4 process of the actual translation of a high-level design into programming code, but extending our investigation slightly into the stage-5 activities of debugging and testing. As a result of our findings concerning stage-4 programming difficulties, we next concentrated on the initial stages of programming prior to actual coding. In Section 4 we detail our work which focussed on the characteristics of natural language as a medium for expressing and communicating procedural requirements, this corresponding to one form of stage-3 designing. In Section 5 we present our study of design as a problem-solving process, in which we investigated the stage 1 to 2 process of problem-formulation and problem-refinement, and the stage 2 to 3 process of creating high-level designs from functional specifications. Section 6 describes our research into the problems of modifying programs. Program-modification corresponds to the stage-6 process of altering the program at a later time. Section 7 deals once again with the broader programming environment, this time focussing on those aspects of computer systems which influence the effectiveness of user performance.

We have attempted throughout the report to provide original theoretical interpretations of the behavior investigated. Much of this theory is presented here for the first time and has still to be tested. Nevertheless, in view of the dearth of psychological theory for almost all of the areas investigated, we felt it would be useful to provide at least a starting point for more adequate theory development. (The most extensive presentations of new theory are found in Section 6, for *program modification*, and in Section 7, for *routine tasks* and for *problem-solving tasks*).

In this report several conventions of referencing are observed. References to the technical reports describing the original work are given as a number in parentheses throughout, and always at the end of major sections. These reports are listed in Section 8. Personnel involved in the contract work are listed in Section 9); the reader can identify the contributors of the research from the author information given there. In all cases the first author of the reports bore the major responsibility for the research work. For simplicity, "we", "our", and "us" are used throughout. Where this is used in reference to the experimental details and findings, the original authors should be credited; indeed, some of this material may be a close paraphrase of the original report. Additional integrative material supplied in this final report -- e.g., research motivation, new theory, summary interpretations, recommendations --is the responsibility of the present author, who, hopefully, reflects the consensus of opinion of the sixteen other contributors. References to published research literature other than our contract reports are *not* given in the discussion but may be obtained by reference to the original reports.

2. *STATISTICS CONCERNING USAGE OF INTERACTIVE COMPUTERS*

2.1 *MOTIVATION:*

The quantitative description of the use of existing interactive computer systems was believed to be a starting point for determining which features of computer systems should be retained, deleted, or improved.

2.2 *SUMMARY OF WORK:*

The source for the data analyses was a 21-day record of user-system interactions of the IBM Watson Research Center TSS/360 interactive system, with a total user population of 375 persons and a total recorded user-session length of 3712 terminal hours. The results are grouped into five major categories of observations: terminal usage patterns, language processors, command usage, user response time, and debugging.

Analyses of *terminal usage patterns* revealed that only 20 percent of the users were high frequency users --signing on to the system daily and accounting for about 60 percent of the total connect time, with very long individual terminal sessions. The majority of the users signed on infrequently, for very short terminal sessions (1).

Three *language processors* were supported by the TSS system: PL/I, FORTRAN, and Assembler. Only 34 percent of the user population used one or more of these language processors (PL/I and FORTRAN were used equally often and about twice as frequently as Assembler). Detected syntactic errors in programs submitted to these language processors ranged from 12 to 15 percent, suggesting a very high level of syntactic correctness (1).

The most significant finding from the *command usage* analyses was that text-editing commands accounted for more than three-quarters of all the commands issued to TSS (1).

User response time was defined as the elapsed open-keyboard time before the user responded, with the overall mean being 32 seconds (eliminating response times over 10 minutes), the median being about 9 seconds, the mode 4 seconds (1).

Analyses of *debugging* behavior were based on usage of specialized TSS debugging commands (the Program Control System, or "PCS", statements) and questionnaire responses from users who employed the debugging commands. It appeared that these commands were not used with any frequency, and, when they were used it was more to control the processing characteristics of the program than for debugging (3).

2.3 *RECOMMENDATIONS:*

2.3.1 *Terminal Usage Patterns and User Response Time:* Different groups of computer users have extensively different profiles of computer usage, characterized primarily in terms of a sophisticated/frequent vs. unsophisticated/infrequent dichotomy. The two poles may also

differ with respect to their needs and expectations, with the sophisticated user probably requiring much more function, performance, and flexibility than the unsophisticated group. This latter group, on the other hand, may be expected to require more assistance, more direction, and greater "error-tolerance" than the former.

In view of these expectations, we recommend testing of a facility which tailors the performance, function, and user-assistance levels of an interactive system to the use-profile of the user to achieve not only greater individualized user support but also more efficient overall computer system performance.

2.3.2 *Language Processors and Debugging Facilities:* The very low level of occurrence of syntactic errors in programs suggest that the development of more powerful syntax checking facilities would not be justifiable. However, facilities should be developed to assist in the detection of *conceptual errors*, errors which, though syntactically correct, produce unintended processing effects.

We recommend that "semantic checking" facilities be developed for use within language processors to aid in the detection of these conceptual errors, parallelling the facilities for detecting syntactic errors. We believe that very substantial aid would be provided by very low-level, easy-to-implement semantic consistency checking which would check for operator-operand consistency/compatibility. With such a facility program variables would be declared in terms of a very rich set of data types corresponding to the semantic aspects of the program -- e.g., in a payroll application, variables would be characterized as to "salary", "date", "contribution", etc. The programmer would further include information concerning allowable -- "meaningful" -- transformations on such typed variables, such as "salary x tax-constant = tax-deduction", etc. The compiler would be extended to perform a data-type analysis of the results of each processing step, matching the result against the set of those allowable, reporting to the programmer occurrences of disallowed combinations. Our experience from this as well as other aspects of our contract work indicates that such consistency checks could lead to the detection of a substantial number of, often hard to detect, conceptual errors.

2.3.3 *Command Usage* : The temporal and frequency analyses of user-issued commands within terminal sessions provided a very rich and informative body of data concerning usage of interactive systems. The disadvantage of the techniques we used was that we had no information to permit us to segregate sequences of users' commands into groups related to particular tasks, nor did we have direct knowledge of what tasks users might be engaged in.

We therefore recommend that it would be useful to develop and utilize software facilities which could be invoked during any individual user terminal session to record and analyze the commands issued by the user for particular tasks (the beginning and end of which the user would indicate). Comparisons of results across users for the same tasks could reveal much finer details of usage related to specific tasks and permit identification of problem areas and potential aids.

We have developed an exemplary software package appropriate for IBM 370/CMS systems which classify a user's commands into several CMS command categories, with additional subcategories provided for commands issued within the supported text/program editors; frequency and time statistics are then provided for the individual commands, for the commands grouped into classes and into pair and triple sequences of commands (19).

3. PERFORMANCE WITH PROGRAMMING CONTROL STRUCTURES

3.1 *MOTIVATION:*

One of the most important characteristics of computer programs -- or any sequential procedure -- is the means for controlling the sequence of execution of program commands, the so-called "transfer-of-control structure." In view of the attention control structures were beginning to receive in the Computer Sciences literature, we believed it important in our first controlled experiments to obtain some assessment of the difficulty experienced by programmers, particularly novice ones, in the specification of such structures.

3.2 *SUMMARY OF WORK:*

We conducted three experiments in this area, the first exploratory, the second a comparative evaluation of existing control structures, and the third an initial evaluation of a new control structure designed to improve performance.

3.2.1 *Initial experiment* -- In our first study we created a simple laboratory programming language for card-sorting tasks which permitted participants to create programs on an interactive system merely by selecting the desired command from a short list of fixed commands, adding only transfer-of-control information. The control structure of this language was that of "branch-to-label". The six card-sorting problems for which participants had to prepare programs differed in logical complexity and involved four types of activities: initialization of data structures, data-accessing, updating counters, and specifying branch-destinations for test outcomes. Despite this opportunity for a diversity of errors, almost all of them occurred with the specification of the control information (with disjunction and negation providing more difficulty than conjunction and affirmative tests). Thus, it appeared that transfer-of-control specifications were the primary locus of programming difficulty. In addition, the debugging of incorrect control-flow programs -- for, usually, one or two statements errors --required half as much time again as for preparing the initial program, with over three times as many editing modifications (2).

3.2.2 *Evaluation of control structures* -- Having determined control specification as a high-probability difficulty factor, in the next study we evaluated several methods for representing transfer-of-control, looking for the best candidate for, in particular, novice programmers. In addition to the "branch-to-label" (BRANCH) structure used in the first experiment we selected for testing two other structures: the "If-then-else" (IF) hierarchical structure, and the graphical flow-diagramming technique (FLOW). These three types of control structures -- BRANCH, IF, and FLOW -- account for the majority of types of control structures used in programming languages.

Although BRANCH and FLOW techniques appear radically different, they are logically equivalent with the differences being in *syntax:* (a) whereas BRANCH programs are one-dimensional lists, FLOW programs are two-dimensional; and (b) whereas transfer-of-control is specified in BRANCH programs *symbolically* by means of a command number or label, FLOW programs involve a *graphical* specification -- i.e., with a connecting line. Performance differences between FLOW and BRANCH could therefore be attributed to one or both of these syntactic differences. Performance differences between either of these techniques and IF, however, must involve the underlying conceptual differences between them.

To provide for a variety of comparison conditions the three methods were each tested under four language conditions in which the capability to employ directly the logical operators of "and" and "or" as one factor and "not" as a second factor were separately manipulated. Parallel miniature languages and training programs were prepared for each method followed by a set of six experimental problems within each condition for each subject.

Results and Discussion -- In contrast to the "logical" arguments being made in the computer sciences literature in favor of the IF structure, we found that this was the most difficult technique to learn and to use correctly. Also suprisingly, there were no differences, either in learning or in performance, between BRANCH and FLOW, both being superior to IF, but still involving control errors in the problems. Finally, the type of language condition had no effect. These results, originally obtained for naive college participants, were replicated for U.S. Navy enlisted personnel (7).

3.2.3 *Procedure table experiment* -- In our third study of control structure performance we developed a new transfer-of-control specification structure which we termed a "procedure table." In contrast to the unstructured format of typical programs, the PROCEDURE TABLE provided considerable structure for the location of information (see example below). Program information was entered into a table which contained columns for specifying the set of conditions which had to be satisfied and for the set of actions subsequently to be taken. When the conditions in a particular line of the table were not met, control was passed immediately to the next line in the table, etc. Two additional columns were provided in the table for command labels and "go-to's" such that, in this respect, the table was most similar to the BRANCH structure. The structure is somewhat similar to a (90-degree rotated) decision table and did, in fact, result from pilot studies of performance using decision tables. In this pilot work we found that the structure of most programming problems we considered involved an "ordering" -- on priority, cost, etc. -- of conditions or subproblems and are more conveniently solved using a programming language which can directly and easily reflect these orderings; we further found that participants experienced difficulty in using decision tables, both in the initial abstraction of the separate sets of tests and actions that should be entered and also, especially, in specifying sufficient test patterns to completely cover all possibilities. Since we were interested in developing a technique which participants found easy to use, we did not further test decision-table performance in formal studies. Rather, we synthesized the PROCEDURE TABLE from our observations of participants' problems.

The PROCEDURE TABLE is illustrated below for a simple-minded "heating" problem and solution.

LABEL	CONDITION(S)	ACTION(S)	GOTO
A1	TEMP TOO LOW?	PUSH "HEATER ON" AND WAIT 1 MINUTE	A1
	- - -	PUSH "HEATER OFF" AND WAIT 2 MINUTES	A1

Parallel language and training materials were developed for this table technique, which was then evaluated under the same experimental design, with the same problems and subject populations as for the prior evaluation. Performance was found to be significantly superior to that with the other types of control structures, with perfect performance found in almost all conditions over all problems (again, no differences were found attributable to the language

conditions). Apparently, the provision of this tabular structure was of key importance in assisting performance (7).

The procedure tables were also used in subsequent pilot work with the two populations exploring performance for much more difficult problems than used in the control structure evaluations -- e.g., programming a "bubble-sort" sorting routine, tic-tac-toe, a program to draw different size boxes on a display, etc. These problems turned out to be much too difficult for these programming-naive populations, with very few correct or even near-correct programs. Informal analysis of the solutions suggested that participants' difficulties were not related to use of the procedure tables: what small amount of detail participants did provide in the tables was pretty much correct, even fairly complicated control information. Subjects' difficulties seemed rather tied to inadequate formulations of the problem and incomplete or faulty designs. These and similar observations from other work led us later to investigate more closely the pre-programming phase of *program design* (see section 4 below).

3.3 *RECOMMENDATIONS:*

Representing transfer-of-control was definitely a source of difficulty in our experiments. However, the effect of providing *structure* for organizing program information was dramatic, both in the improvement in performance and also in the ease of learning and the participants' very positive attitudes towards the technique. In view of the power of present-day compilers to produce very efficient programs from inefficient code, we recommend that a structured technique like the Procedure Table be used for program coding under the following four conditions:

(a) the program is for "stand-alone" use, not involving complicated interrelationships with other programs, as in systems programming or other real-time processing applications with interrupts and complicated calling structures;

(b) the application for which the program is written is well-understood and highly standardized, there being a stable vocabulary of terms, descriptions, tests, etc.;

(c) the data structures required by the problems are fixed and are relatively simple -- e.g., push-down stacks, arrays, etc. -- and they also do not require complex dynamic manipulation (pointer variables and dynamically modified linked lists thus should not be required);

(d) the problems to be solved by the programs are relatively easy to comprehend, not requiring creation of some complex or highly innovative algorithm.

We estimate that a large proportion of the programming tasks in businesses, governmental services, and the armed forces, would meet these four conditions.

Given the above we would expect that personnel with little programming instruction could produce correct programs quickly and easily for a wide variety of specialized application problems without requiring the assistance of experienced programmers.

Productivity could further be enhanced by providing a "menu" of conditions and actions from which to select for entry on pre-established forms (either paper or computer-display). Such a pre-defined and limited vocabulary would facilitate implementation of the "data-type" characterization recommended in Section 2.3.2. With such facilities programmers could be

informed not only of the use of unrecognized words, but also of low-level "conceptual" errors violating pre-defined semantic relations. We estimate that the development of software to support interactively such features, including translation of the input structure into some formal programming language, is only a moderately difficult task, far easier than most compiler-writing activities.

4. NATURAL LANGUAGE PROGRAMMING AND COMMUNICATION

4.1 *MOTIVATION:*

Subjects in our programming experiments often complained that programming languages were difficult to use and were "unnatural"; it would be much easier to give solutions "*in* English", they said. Given the ubiquitousness of natural language procedures, and their apparent successful communication of information, we decided to assess the extent to which natural procedures *did* differ from formal programs -- in style, precision, and principle of communication.

4.2 *SUMMARY OF WORK:*

Our research in this area followed three different lines of inquiry. The first and major investigation, Section 4.2.1, concerned how process information was conveyed in natural language procedures. The second aspect of this work, Section 4.2.2, involved an investigation into natural language queries as a non-procedural alternative to programs (natural language or otherwise). The third aspect, Section 4.2.3, provides a broader perspective on the processes underlying natural language communication. Each of these is discussed separately in the subsections below.

4.2.1 *Natural Language Procedures:*

Two experimental topics are discussed in this section: (1) Collection of initial procedural data, and (2) Professional natural language programming.

4.2.1.1 *Collection of initial procedural data* -- Our initial experiment was designed to provide data for contrasting computer programs to natural language procedures written for similar purposes. College students were asked to type detailed specifications of procedures in natural English as solutions for a set of six file-manipulation problems. These problems varied in complexity and involved the scenario of retrieving employee information from personnel files of a hypothetical company. The resulting written language productions were examined from the points of view of solution correctness (e.g., generally correct), preferences of expression (e.g., much more concerned with *data* manipulations than with control factors, preference for aggregate data operations rather than iterative -- e.g., "Find *all* those who ..."), contextual referencing (e.g., 42 percent of all data references required prior context for resolution of the referrant), and word usage (e.g., relatively small commonly-shared vocabulary). A table from the report (5, Table 6) is adapted and shown below to provide a summary of the detailed contrasts of natural vs. programming specifications.

Characteristics of Typical Programming Languages vs. Natural Language Expressions of Procedures

FEATURES	PROGRAMMING LANGUAGE	NATURAL LANGUAGE
DATA ASPECTS		
References	Explicit, well-defined	Implicit, contextual
Manipulation	Element by element	On aggregate basis
Indexing	Frequent, by variable or integer value	Rare, then relatively, e.g., "next, last"
Types	Many, pre-defined	No distinctions
TRANSFER OF CONTROL ASPECTS		
Extent	Major program aspect	Seldom specified
IF-THEN-ELSE	Common feature	Rare
Branching	Very common	Never occurred
Exception-handling	Important aspect of most programs	Never occurred
Program flow	All varieties	Basically linear flow
LEXICON/SYNTAX		
Lexicon	Restricted words and variable names often in range of 100-200.	About 800 words, reducible to 100-200 with elimination of synonyms.
Sentence Form	Primarily imperative	Wide variety of forms but imperative predominant.

At a more general level, the most dramatic finding concerned the *style* of the natural language vs. programming procedures. Programming languages embed the processing action deep within control structures that test for a variety of successive conditions. Thus, given the problem of writing a procedure to describe the packing of 200 boxes of a certain color and shaped Christmas tree decoration, unbroken, into boxes, with a dozen in each box, during working hours, a (somewhat exaggerated) programming-style solution might be as follows:

```
UNTIL TIME = 5:00 PM
IF RED THEN
   IF LARGE THEN
      IF UNBROKEN THEN
         DO END1 J=1,200
            OPEN BOX(J)
               DO END2  I=1,12
  ----------> PACK DECORATION(I) IN BOX(J)
               END2;
            CLOSE BOX(J)
            END1;
      ELSE RETURN;
   ELSE RETURN;
ELSE RETURN;
```

Notice that the primary action of packing, as indicated by the arrow, is at the deepest level within the pseudo-program. This style may be called *conditionalized action* in view of the surrounding of the action(s) by sets of conditions.

In contrast, natural language procedures emphasize the action rather than the conditions governing the action. These latter are expressed as verbal qualifications, either on the object of the action or on the action itself. Thus, the above procedure could be expressed naturally as:

-------> PACK LARGE RED DECORATIONS TWELVE TO A BOX.
MAKE UP A TOTAL OF 200 BOXES.
STOP AT 5:00 PM IF NOT FINISHED.
BE SURE TO PACK ONLY THE UNBROKEN ONES.

Here, the packing action, again indicated by an arrow, is the very first item of information. In general, in natural-style procedures, the actions are stated first, with what was the conditional information now expressed as qualifications. These qualifications occur within the noun phrases (e.g., "... *large red* decorations ..."), or as modifiers of the verb phrase (e.g., "... *twelve to a box*", which qualifies the action of packing). It is because of this qualificational aspect that the natural style is characterized as *action qualification*.

The natural and programming styles differ not only in the characteristics as highlighted above but also in a number of other ways -- e.g., the ordering of the particular information, the presumed default actions, syntactical constraints, type and arrangement of operands, etc. Thus, the differences are not just superficial, but profound (7).

4.2.1.2 *Professional Natural Language Programming* -- Given the prevalence and understandability of natural language procedures --e.g., assembly instructions, repair manuals, kitchen recipes, knitting instructions, etc. -- and given the difficulties experienced in using programming languages, we decided to further pursue our investigation into natural language procedures. Our objective was to determine the extent to which natural procedural communication could provide an alternative model for designing the language interface between computers and the general population. We sought to identify the underlying *mechanisms/processes/conventions* governing the communication of procedural information from a written text to a human reader.

After analysis of a number of procedural texts, we have made significant progress in developing an overall model of some primary features of natural language procedural communication. It appears that the imperative verbs of natural language procedures are to be understood not as primitive actions but as a programming-like "calls" to some underlying *procedures*. These underlying procedures are usefully conceptualized as having four components: (a) a set of preconditions to be satisfied before execution of the procedure, (b) a "program" of (decompositionally more primitive) actions sequentially organized to capture the "essence" of the particular procedure, (c) a set of consequent conditions resulting from execution of the procedure, and, (d), a set of "operands" or "arguments" characterizing the various classes or (semantic) *cases* of information that could or must be supplied to the procedure program. This latter component may be called a "case frame" and is critical to the appropriate semantic interpretation of the surface text information.

Given this model, the understanding of natural language procedural units (typically, independent clauses) is thus presumed to involve two kinds of coordinated mapping between the surface linguistic information and some internal "understanding" representation. First, a mapping is established between the surface verb and the appropriate underlying "procedure", which process may well be guided by a number of other factors including prior linguistic context, the present "state of the world", and the other linguistic information co-occurring in

the same sentence. Second, segments of the surface text are mapped into the various case categories made available when the first mapping returns the associated case frame. Again, a number of factors influence this process (7, 13).

Preliminary consideration of a wide variety of different types of natural language procedures suggests that not only does this model seem useful for the variety of application domains but also there is considerable commonality in the specific case-types across domains. Further, this commonality presently seems to extend even to much more detailed mechanisms, such as the subtle use of the *position* of a past participle relative to the noun it modifies to indicate when a particular action should be or should have been accomplished. For example, in the popular domain of cooking, the pre-nominal position of "buttered" in "Serve the buttered toast" implies that the buttering was accomplished some time before this point; however, the post-nominal position in "Serve the toast buttered" implies that buttering is to occur just prior to the serving (13). As a sidelight, it should be noted that the *pragmatic* nature of our analyses -- investigating goal-oriented effectiveness-minded activities -- has given us a kind of "leverage" for a deeper analysis of some traditional linguistic problems, such as coordinate conjunction (11).

In other related exploratory experiments, computer-naive participants were asked to write and carry out natural language procedures, natural language descriptions, and instructions expressed in an artificial restricted-syntax language. Writing performance -- speed and accuracy -- were about the same for all three approaches, although the linguistic characteristics differed greatly from approach to approach. While participants were tolerant of ambiguity both in writing and in carrying out instructions, they often voluntarily employed restricted-syntax notations in their writing after being exposed to the notation (10). This last finding, together with the previous finding suggesting that participants could comfortably specify procedures with a drastically reduced vocabulary (5), implies that a language interface based on natural but considerably restricted syntax and lexicon (with suitable data-type semantic checking, as discussed elsewhere) might well be an acceptable and preferred alternative to the present programming-like interfaces.

4.2.2 *Natural Language Queries:*

While the above-described work constituted our major effort in investigating natural language communication, we also conducted some exploratory studies concerning other related issues, as described in this and the next section.

The present section relates to programming applications whose objective is simply to retrieve selected information from some data base. Given a formatted data base, a program can always be written to produce an output of information based on any combination of selected information attributes. However, there are several kinds of so-called "query systems" which permit entry of the information request in much more natural-like terms without the requirement of writing a program to accomplish the information retrieval. For example, a user might enter "who are the managers" or, equivalently, "list all the managers", and the underlying query system would automatically translate this information into a program which would in turn access the data base and return the information. Such queries are presumably much easier for users to formulate than programs, and thus they represent a non-procedural alternative to programming. It is this last characteristic which motivated our interest in queries and query systems under the contract.

Three aspects of query situations can be identified: (a) *the information problem* -- what information is required and why, (b) *the information source* -- the structure and contents of the data base (presumably) containing information relevant to the problem, and (c) *the information*

language -- the nature of the communication language(s) in which the user expresses a query and in which the computer system supplies information to the user.

The three experimental topics discussed in this section are: (1) Behavioral performance with data structures, (2) Use of a query language, and (3) Quantification in queries.

4.2.2.1 *Behavioral performance with data structures* -- In our first study we were concerned with the second of the above aspects, the information source. In particular, we were concerned about the congruity or compatibility between the user's "view" of how data should be organized and the actual data structures within the computer system. Discrepancies could lead to various kinds of difficulties in satisfying the information problem.

Our experimental approach was to ask participants to organize lists of words into various kinds of structures under the following conditions: (a) the word lists always had some pre-defined inherent organizational structure which was more appropriate than any other; (b) the organizational structures used were hierarchies, networks, lists, and tables (with certain modified forms of these in addition); (c) special "skeletal structures" giving the form in which the words should be organized were sometimes provided, and (d) participants were sometimes asked to organize words into specific structures as opposed to choosing any structure they felt best.

Results and Discussion -- One primary finding of the experiments was that, over a variety of conditions, participants conceptualized data most easily in terms of lists and with most difficulty in terms of networks. Performance with tables was moderately good even though the word groups with pre-defined tabular structure were not so organized as to constrain other forms of organization. A second important finding was that participants were able to discover the semantic relations among the words and select an appropriate organizational structure for best representing these relations. In addition, the providing of skeletal outlines for structuring the words did facilitate performance as expected, and participants had difficulty in forcing data into inappropriate organizational structures.

These results have several implications for the nature of data structures in information retrieval systems. First, it seems very likely that performance will be affected by the manner in which the data organization is represented to the users; this factor must be taken into consideration. Second, the conceptual structure of the data base should conform to the semantic relationships among the data elements. Third, the language used for interrogating the data base should have terms and relations in it appropriate to the structural view of the data given the users. Finally, in the entry of data into existing structures, providing users skeletal structures to guide the entry format should facilitate performance (4).

4.2.2.2 *Use of a Query Language* -- A second, exploratory, experiment concerned the third aspect of query situations given in 4.2.2 above, the information language, in which we studied the problems of translating from some information requirement into a specific query format. We examined separately the formulation, planning, and coding of queries by college students and older, less educated, file clerks, using a query language somewhat similar to IBM's IQF query language but containing more function.

Results and Discussion -- The form of the information problem given to participants, well-expressed or diffusely expressed, seemed to affect problem formulation time but had no effect on problem planning or problem coding times. Of the 195 queries written, 127 were incorrect. A primary source of error was the incorrect specification of the value of the

attribute, particularly for ordinal relationships, such as "age over 50", which had to be translated as "51 or more" in the query language. Another large source of errors was omission of various values, operators, headings, etc. A third source of error was inappropriate use of the commands for counting and averaging, and the commands for indicating conjunctive or disjunctive relations among parts of the query. We found that the college population performed better in several respects than the file clerk group, both in training and in test performance. Overall, this experiment impressed us with the critical importance of providing features in the query language that were not only sufficient for expression of the queries but were also compatible with "natural" tendencies (6).

4.2.2.3 *Quantification in Queries* -- Our experience with the above and a previous query system indicated that, among other problems, participants often had difficulty in the use of universal quantification in queries -- e.g., the use of the terms "all", "each", "no", as in "Find the salaries of *all* managers over 50 years of age". The data were collected in a variety of procedures using nonprogramming participants. These non-programmers variously translated English questions into a query language, translated Venn diagrams into English or vice versa, gave judgments about the consistency of two English statements, or manually looked up answers to questions.

Results and Discussion -- Subjects showed considerable difficulty with the logician's notations of set relations (except disjunction) on all tasks. The interpretations given quantified sentences varied between participants on a given task and even within a participant between tasks. Generally speaking, participants gave interpretations consistent with quantified natural language questions or Venn diagrams, but not equivalent to them. They rarely used explicit set specifications in spontaneous English.

Summary interpretation of the findings provide for the following tentative recommendations: (a) if a query system forces a user to employ the logician's quantifiers a high proportion of errors should be expected; (b) whenever practical, the system should communicate to the user in terms of set identities and set disjunctions; (c) it is better to let users choose among system-produced quantified descriptions to satisfy their needs than to have users *generate* their descriptions in free-form; (d) a query system should generally not attempt to answer exactly and only the users' precise questions when those questions involve quantifications; rather, the system should expect large differences among users in the use of language and provide more information than requested (8).

4.2.3 *Natural Language Dialogues:*

The previous two sections have been concerned with natural language programming in a non-interactive one-way sense. That is, the natural language message is given to some recipient who then attempts to understand it, without further interaction. Of course, natural communication also occurs most frequently in an interactive or *dialogue* mode, in which two or more parties participate interactively in communicating and understanding the message. One can imagine, as an alternative to programming, a dialogue with an interactive system in which the user is prompted for information concerning his task, input information, clarifications, resolution of inconsistencies, etc. The structure of the information flow in such an exchange can be anticipated to be considerably different from the imperative passing of a completed instructional message, and we were interested in determining what some of the characteristics of dialogue situations might be. In addition, we were interested in developing a better theoretical communication model of information transfer in dialogues.

4.2.3.1 *Dialogue Methodology* -- From our intial exploration into dialogues we developed an experimental technique for studying application-specific dialogues, in which a user interacts via typed messages with a second person who is simulating a computerized natural language interface. Although our efforts were largely exploratory, we do feel some useful tentative generalizations can be made. First, in natural language dialogue situations an important aspect of the interaction is not task-specific but concerns the communication process itself, as when participants comment on the communication channel, on the clarity of the information, etc. Second, the way in which various expressions (e.g., conditionals) are used depends heavily on the pragmatics of the dialogue -- the communicant's task objectives -- not just the isolated semantics of the words. Third, users with different backgrounds will interact quite differently with a natural language interface, being more or less able to realize the constraints and limitations of such an interface (9).

4.2.3.2 *An Alternative Theory of Dialogue Communication* -- Most views of human-computer interfaces, and, indeed, of human-human communication, include the imlicit belief that communication from system A to system B essentially involves the following: Some internal state in system A is coded into an external statement for transmission to system B; system B decodes this message and changes its internal state accordingly; communication is considered "good" to the extent that there is an isomorphism between the internal states of the two systems after the message has been sent.

Empirical results as well as theoretical grounds provide a basis for arguing that this view is inadequate both for an understanding of communication between two persons and as a theoretical foundation for any kind of human-computer interaction. Instead, an alternative model is proposed which stresses the game-theoretic aspects of communication, the importance of viewing message-building as a *constructive* rather than a translational process, the importance of "metacomments", and the multiplicity of channels involved in natural language communication. In addition, this view argues that, under certain conditions, the "vagueness", "fuzziness", and ambiguity of natural language can be assets not liabilities (15).

4.3 *RECOMMENDATIONS:*

We come away from our research in natural language programming with the very strong impression that when the communication is for some definite purpose natural language can be remarkably precise, unambiguous, and comfortingly redundant and supportive. (However, the mechanisms by which all this is accomplished are very complex and have yet to be unraveled). We also have the strong conviction that the underlying mechanisms of natural language communication will provide the best models for designing optimal language interfaces between computers and people-at-large. This conviction is based on the beliefs that user-computer communication modelled on natural language communication: (a) would be the easiest to learn and to teach; (b) would provide for the greatest "richness" or density of information per unit interaction; (c) would provide the widest range and most flexible of mechanisms for improving communication efficiency -- including mechanisms for preserving the context of discussion, permitting "shorthand" abbreviations and omissions, for extending or modifying communication concepts or modes, and for considering widely differing topics and data-forms in the same discussion; and (d) would provide the most sophisticated and powerful techniques for recovery under communication "noise" or error.

In order to develop the necessary models of natural language communication much basic behavioral and psycho-linguistic research is needed. Our recommendations for the area of natural language programming and communication focus on four such areas: Lexicon and

syntax, text cohesion, practical knowledge structures, and descriptions (the previous sections in 4.2 describing work accomplished already provide some topic-specific recommendations).

We believe that most of the information required for determining the underlying communication mechanisms of natural language could be obtained by studying purposive procedural communication (e.g., operating instructions, assembly directions, etc.) in the form of written text or interactive dialogue.

4.3.1 *Lexicon and Syntax* -- There are many practical difficulties in the way of natural language user-computer communication, even restricting the dialogue to specific tasks. A number of these could be reduced or eliminated if the interaction could be further constrained in various ways. Two of the most powerful types of potential constraints concern (a) the size of the recognized lexicon or dictionary, and (b) the set of recognized grammatical constructions. We recommend that empirical studies be conducted to determine the regularities that characterize purposive procedural natural-language communication. These data would then serve as a basis for suggesting (and subsequently evaluating) possible constraints.

More specifically, with respect to lexicon size, the vocabulary of texts and dialogues should be characterized as to part of speech, with subsequent analyses focussing on such things as abbreviations and short-forms (e.g., "Ms" for "manuscript", "auto" for "automobile"), synonyms (e.g., "book, paperback, novel"), or morpheme-stem variations (such as endings for number -- e.g., "book, books" -- or stem suffixes producing additional words within the same or different classes -- e.g., as the words "care, careful, carefully, caring" involve the same morphological stem). These categorical analyses could provide a basis for determining minimum vocabulary classes and efficient lexicon-construction practices.

With respect to syntactic structures, procedural texts (and dialogues) should be analyzed for syntactic surface structure characteristics as well as for usage of linguistic devices which lead to cohesion of the text. The structural analysis should have particular sensitivity to variations in noun phrase constructions, as our experience indicates that natural language procedures "package" an extensive amount of process information within the wide diversity of these structures.

4.3.2 *Text Cohesion* -- Procedural texts and dialogues should also be examined for the linguistic mechanisms by which separate segments are connected or made cohesive. This analysis should include descriptions of at least the following three aspects The first, (a), is *anaphora*, the resolution of noun phrase references either to prior text entities or to entities outside of the text, inferrable from the procedural goals, etc.; in addition to the use of pronouns, the use of articles (e.g., "a, the") and verbal quantifiers (e.g., "one, each") in anaphora should also be investigated. Two additional cohesive techniques, whose usage should be described are (b) *ellipsis*, the omission of certain textual information which, nevertheless, the recipient of the text can recover by various means, and (c) *substitution*, the replacement of words, not by pronouns or demonstrative adjectives (e.g., "this, that"), but by synonyms. Empirical descriptions of these "text-binding" processes would provide important information for the determination of the level of complexity of linguistic-processing mechanisms required in a user-computer interface to support the cohesion or "context-forwarding" of extended discourse or dialogue.

4.3.3 *Practical Knowledge Structures* -- Once a computer language-interface departs from a fixed set of pre-defined commands with pre-set operand formats, there must necessarily be an interpretive process of assigning "meaning" to the symbol string input by the user, and this is required whether the language is a natural language or some more flexible programming language. While a parsing process using a set of grammar rules and at least a part-of-speech

lexicon is certainly required, this is insufficient for the assignment of "meaning" to the input. This latter process requires the discovery of the conceptual relationships among the various components of the given input string, as well as the relationships to prior input material or to the user's task, the state of the "world", etc. Information concerning various possible conceptual relationships must also be available to the processing systems for determination of the specific concepts conveyed by any particular input. The repository of such information can be called a "knowledge structure", and many questions regarding the nature of human "knowledge structures" can be raised.

One of the most important of these questions for computational linguistics is whether there are relatively independent and separate knowledge structures associated with different task domains, or whether all knowledge from all domains is inextricably intertwined. The latter possibility implies that every meaning-interpretation activity within some particular task domain may, and possibly must, require access of broad knowledge outside of the specific activity domain. Such a view would be discouraging for projects whose goal it is to support natural language communication process within some limited practical domain. What needs to be experimentally determined is the extent to which human meaning-interpretations within such domains can be accomplished with limited knowledge structures containing only the practical and highly relevant body of conceptual information related to the particular domain. Is it possible, for example, for human participants to supply a relatively small and well-delineated set of conceptual relationships for some limited domain which would constitute "all one needs to know" to accomplish the majority of normal activities within that domain (e.g., what is there to "know" about sending a telegram, signalling a ship, checking inventory, etc.). Further, what is the structural characteristic of such information-- e.g., how are associations accomplished, what is the complexity of the relationships, how is information search mediated, etc. Certainly, as interesting as these problems, are other questions concerning the *dynamic* characteristics of knowledge structures, such as the building of internal "world-model" representations reflecting the changes produced by successive sequential action-events. We believe it is possible to obtain such information in the behavioral laboratory, and we further anticipate that relatively independent knowledge structures *can* be inferred as mediating the practical domain meaning-interpretations, thus opening the way for intensive modelling of these processes as a guide for ultimate computer implementations.

4.3.4 *Descriptions* -- The final research area we recommend as instrumental for achieving eventual natural language processing by computer concerns the seemingly more narrow area of natural language descriptions. We propose that research be conducted to determine the conventions and considerations governing descriptions within the context of procedural tasks. We believe that such information not only would be enormously useful in developing powerful natural language systems, but also this knowledge could be employed in a variety of other computer-supported areas to significantly improve the power and "naturalness" of the application interface.

We distinguish two forms of natural language descriptions: *naming* and *characterization*. In *naming*, some set of symbols is assigned to denote some particular constellation of actions, relations, entities, or events among which there is, at least to those who name, some perceptible form of association which permits and, perhaps, governs, their act of identification. Thus, naming involves a "stands-for" or "invokes" or "calls-to-mind" relation between the symbol name and the referent constellation.

The motivations underlying naming must certainly include the following purposes of: (a) discrimination of one constellation of items from others (as when a particular puppy in a pet store is referred to as "the cute white one with the bent ears"), (b) facilitation of subsequent communication (as in legal documents where such terms as "the first party", "the defendant", etc. replace specific "real" names), (c) summarization of features for the communication

purposes of the moment (as a conservative politician's warlike bias might be captured in a reference such as "This unremitting hawk ..."), and, particularly in *procedural* communication, (d) the deliberate compaction, for the sake perhaps of brevity, of a variety of descriptive and process information into a single syntactic unit, e.g., noun phrase (as in a kitchen recipe the requisite pre-processing of an ingredient is conveyed by the phrase "one-half chopped green pepper, seeds and membrane removed ...").

The experimental revelation of what must be the extremely complex, multi-purpose, and multi-level nature of naming would be highly significant even if used only for the purpose of providing sufficiently powerful and flexible facilities in computer systems to support natural language naming habits. In addition, of course, a determination of the underlying process of naming would also be most revealing of the other process of complex natural language communication -- e.g., dynamic "world-modelling", nature of static knowledge structures, "meaning" derivations, etc.

The other facet of natural language descriptions, *characterization*, refers to the purposive selection of a subset of features from a nameable constellation of entity-action-events, such that this subset, typically, fulfills an information request -- present or anticipated -- about the particular constellation under consideration. Thus, a geologist runs through a series of chemical tests on a moon-rock (named, e.g.,"Sample 24667") to obtain a set of measurements which permit inferences concerning the molecular-ancestry of the rock; in contrast, a UFO believer holds sample 24667 up to the light searching for some sign of non-natural processing or imprint.

We conjecture that the dominant information requirement motivating *characterization* is that of predicting one particular subset of features of a constellation from another subset. Thus, students' academic potentials are predicted via IQ measures, the quality of a used car is indexed by the result of a tire-kick, etc. Much closer to the area of computer applications is the characterization of a document in terms of "keywords" such that a subsequent information request could lead to retrieval of the document on the basis of a similarity in the information sought and the way the document was characterized.

We believe that an investigation of the processes of natural language characterization would benefit all aspects of computational linguistics and, in addition, could be usefully applied to improving computer capabilities in a variety of areas. For example, for the above-mentioned area of information retrieval the provision of sophisticated natural characterization facilities could greatly facilitate the scientists' problems of appropriately describing documents for filing as well as for subsequently locating these and other relevant materials at a later time. This is becoming a particularly acute problem given the burgeoning increase of information as well as the true cost of failing to obtain the needed data.

5. *INVESTIGATION OF PROGRAM DESIGN BEHAVIOR*

5.1 *MOTIVATION:*

Our experience with the problems of programming performance (see Section 3) indicates that poor quality software can often arise from problems in the *design* of the system rather than in the *coding* of the design into a programming language. Certainly, quantitative figures indicate that software design accounts for a major portion of the costs of overall software development, and much of the problems in software delivered to customers can be traced back to inadequate design (see 17). Having previously investigated some of the problems in translating high-level designs into programming code, we now turned to the more complex problem of achieving the initial designs themselves.

We view design as being one of the most complex types of problem-solving task (see Section 7.2.3.2). And, like most complex real-world problem-solving, there are almost no behavioral studies of design -- and very little theory. While there were a number of competing software design methodologies known to us, purporting to result ultimately in superior programs (e.g., "top-down structured" design, "modular" design), there was very little case-study information on which to evaluate their liklihood of success. Further, all of these methodologies depended on a complex interrelationship of a number of different factors, such that it would be extremely difficult to evaluate separately any of their components. In addition, this complexity led to the requirement of weeks or months of training, just to achieve initial competence in the method. Finally, most of these methodologies were intended for large programming projects involving many software professionals over many months, or even years.

All of these factors made it inadvisable for us to begin our investigation of designing by undertaking comparative evaluation of the various design methodologies in the experimental laboratory. We believed that fundamental investigations of the design process itself would best provide the empirical base of knowledge for subsequent identification of useful design methodologies. Accordingly, we used our knowledge of the software development process to identify important component processes, and we then studied these processes in a variety of ways in laboratory experiments.

5.2 *SUMMARY OF WORK:*

In section 5.2.1 below we present a 4-stage expository model of designing. We then present, in subsections 5.2.2 - 5.2.4, our research work in terms of the model, that relating to the initial, early, and later stages of design, respectively.

5.2.1 *Design-model and Overview:*

Our research is organized within a four-stage model of the design process, beginning when a client first notifies a designer (they may be the same person) of an initial problem and continuing up to the point just prior to the actual construction of a prototype product (e.g., a working program). This model accurately describes the phases of software design, but it is also intended to be able to apply equally well to all other kinds of design activities as well (e.g., the design of buildings, machines, organizations).

that with the other types of control structures, with perfect performance found in almost all conditions over all problems (again, *no differences were found* attributable to the language

The four stages of the model are --

1. *Problem Understanding* -- arriving at a general agreement as to what are: (a) the goal objectives, (b) the systems or environments involved, (c) the constraints (on performance, delivery, cost, etc.), and (d) the resources available to assist in design development (e.g., test-sites, expert advisors, historical data). Both the designer *and* the client may iterate through several revisions in determining what the problem "really is."

2. *Functional Requirements Specification* -- determining precisely what the final product must be like, including: (a) every important aspect of its internal performance, (b) the characteristics of its intended operator and user population, (c) its relationship to other systems and environments, and (d) the development constraints (delivery schedule, costs, nature of acceptance tests, etc.).

3. *Overall high-level design* -- translating the functional requirements into a comprehensive design which specifies the major components of the to-be-developed product, and, for each, describes: (a) the goals to be achieved by that component, (b) the characteristics of all factors to which the component is to be sensitive (e.g., "input"), (c) the characteristics of effects the component must achieve (e.g., "output"), (d) the internal structures of the component (e.g., the internal "data structures"), and (e) the general principles of any operation sequences within the component (e.g., the information processing procedures).

4. *Detailed design suitable for prototype development* -- providing sufficient elaboration of each of the major product components to permit the construction of actual (working) prototypes directly from these detailed descriptions.

We describe our research work below in terms of these stages or in the transitions between them. While the specification of the sequential operations in procedures is a primary component of software design, the design of *structures* is also of critical importance; indeed, some of the design methodologies emphasize the criticality of designing *first* the data structures and then developing the procedures to manipulate the information within them. Accordingly, some of our work investigated the design of structures as well as the differences in design performance for structural vs. procedural types of problems.

5.2.2 *Problem Understanding in the Initial Stage of Design:*

Much anecdotal evidence exists to document the frequency of the following kind of design error: the client describes the design requirements to the designer who, after extensive development effort, produces something that the client doesn't want at all. While it is not clear which party should bear the major blame, it is certain that design solutions produced for ill-formulated or incorrectly-communicated problems will likely be unacceptable to the client.

The primary practical method for insuring correct understanding of the design problem involves extensive discussions between the client and designer to clarify the requirements. Our inquiries and observations suggest that one useful technique employed by designers is to check their understanding of a client's point by suggesting a specific concrete piece of equipment or procedure which might be an approximation to an acceptable solution; very frequently the client rejects a number of these suggestions but modifies the requirement description after each. It appears that the a primary result of this interchange is that the clients clarify *their own* understanding of what their problem is (see 5.2.3 below).

Such a pre-defined and limited vocabulary would facilitate implementation of the "data-type" characterization recommended in Section 2.3.2. With such facilities programmers could be

The research reported in this section focusses on the nature of problem-formulation and possible aids for this process. If clients were able to present better formulations of their problems to designers, not only could the problems of unacceptable designs be reduced but also the design process could be speeded.

The three experimental topics discussed in this section are: (1) Structured problem aid, (2) Unstructured problem aid, and (3) Understanding the problem context.

5.2.2.1 *Structured Problem Aid* -- A review of the literature relevant to problem-formulation (e.g., psychological problem-solving, creativity) revealed quite a few highly-touted *methodologies* for facilitating initial problem-formulation ideation, but very little *empirical* evaluation of the methods. As our first experiment we therefore synthesized from what seemed to us to be the best approaches and created a *structured* aid to facilitate correct problem-formulation.

The aid given participants forced them to focus their attention successively upon the objects and attributes of the information given with the problem and also upon the actions possible within problem-solution. Specifically, we asked participants to answer four questions for each of the four problems given them. These questions were:

1. List all of the objects that are involved in this problem.

2. What can you do to these objects, or what can these objects do to each other? (like "move", "transform", "substitute", or "eliminate").

3. Write down a precise statement of how things are when the problem begins.

4. Write down for every object the characteristics of that object (either by itself or in a relation to other objects) you think might be important in solving the problem.

We chose 5 "puzzle" or "brain-teaser" type problems for testing with two experimental groups, one receiving the aid, and the other given a comparable amount of time to "think" about the problem. These problems were well-constrained and did not require special technical expertise; their solution depended, rather, on achieving the correct abstraction of the problem elements -- the topics addressed by the questionnaire.

Results and Discussion -- We found that the structured aid not only did not facilitate performance, but that performance was better without it. Examination of the responses to the questionnaires indicated that participants had not responded with the levels of abstraction that we had hoped the aid would induce. Rather, their responses were extremely concrete and limited, focussing on *physical* attributes and operations rather than *symbolic* ones. Take, for example, the "four-card" problem: 4 cards known to have a letter on one side and a digit on the other are placed such that the symbols "A D 4 7" are in view, one on each of the four cards; the assertion is made that "if a card has a vowel on one side, then it has an even number on the other side"; the problem is to determine the cards "you would have to turn over to figure out whether the rule is TRUE for these four cards." In response to the first question asking about problem objects, some participants, while mentioning that there were 26 letters and some number of digits, did not even mention the classification of objects into

vowels vs. consonants and even-numbered vs. odd-numbered digits; similarly, for the second question, few participants mentioned any kind of perceptual *testing* as an operation -- e.g., determining if the item was a number or letter, whether odd or vowel, etc. -- and no participant considered the *propositions* of the problem as objects. This problem has previously been used to demonstrate limitations on people's use of (the Piagetian notion of) *formal operations*. However, our interpretation of the present data is that it may not be the case that people are *unable* to perform formal operations; rather, their representation of the elements of the problem may be so concrete that it does not induce them to apply these operations.

In general, we conclude that many of the difficulties experienced by participants in solving our problems derive from fundamental misunderstandings concerning the goals, the initial conditions, and the properties of objects. Our specific aid, although intended to do so, failed to provoke sufficiently abstract examination of the *conceptual* structure of the problem. It may be that the questionnaire instructions lacked the sufficient cues to lead participants to think about the problem in more general and abstract ways; indeed, they may have been cued inadvertently to focus on more concrete aspects by the word-choices in the aid, such as "object", "action", etc.; this could account for the depressed performance with the aid. Whatever the explanation, we were sufficiently impressed by the inadequacy of problem-solving, both with and without the aid, to retain our assessment of problem-formulation as being a critical and error-prone stage warranting further investigation (14).

5.2.2.2 *Unstructured Problem Aid* -- The structured aid in the first experiment derived from the hypothesis that focussing participants' attention on the important aspects of the problem, in a *structured* manner, might facilitate performance. In our second experiment investigating aids for problem-formulation we took a different approach. Perhaps the difficulties of correct problem-formulation derive from viewing the problem aspects too narrowly or emphasizing the irrelevant features -- the problem-solver simply views the problem from the wrong perspective. An appropriate aid, then, would facilitate the solver's breaking of these unproductive modes of thinking by stimulating other avenues of consideration. However, not knowing how the solver was viewing the problem, but knowing that there would probably be wide diversity in approaches, it was unlikely that we could develop a general structured aid which could assist each person to detect each of their varying conceptual "blocks."

One possible solution to this dilemma is to provide a rich variety of cues some of which might stimulate people to identify new and more productive formulations of the problem aspects. Our specific, rather unorthodox, notion was to have people begin thinking about the problem and then have them simply read through a more or less random set of words and phrases taken from a wide number of separate domains of human knowledge. We reasoned that each such item might bring to mind various different semantic concepts and relations, and some of these might prove useful for better formulating the specific problem at hand.

A second set of problems was employed in this experiment with the *unstructured* word-list aid: two "insight" problems similar in structure to those of the first experiment, and two "design"problems (for a structure and a procedure)

Results and Discussion -- While again numerous problem-solving difficulties were found for both the control and the aided-groups, the unstructured aid did appear to facilitate performance for the design problems. Participants' reports of their mental experience while viewing the unstructured word-list aid were consistent with the interpretation that certain of the stimuli provoked productive lines of thought concerning the problem (14).

The findings of these two, admittedly exploratory, experiments holds promise for the development of more effective aids based on *both* the structured and the unstructured approaches, but with greater direction for appropriate abstraction in the former and more problem-related selections of word stimuli in the latter.

5.2.2.3 *Understanding the problem context* -- Our third experiment concerning problem-formulation was tied much more closely to the actual problems experienced by software designers in determining the client's problem. Since most design problems are embedded in complex environments having many constraints which must be observed, we belived that one potential source of difficulty in correctly identifying the problem to be addressed was the correct understanding -- by both the designer *and* the client -- of the nature of this constrained environment.

One of the common design situations requiring precise understanding of the application environment is the following: some procedure which is presently accomplished "by hand" is to be automated. For example, many business procedures can be converted from manual to programmed processing -- e.g., billing, inventory-control, accounts payable, etc. If a software system were to be developed "from scratch" to automate some such procedure in a business, the designers would first have to gain a thorough knowledge of how the processing occurred normally. The present experiment investigated this learning process by asking participants to learn a very realistic 114-operation business procedure for the processing of order invoices. Learning was tested by asking participants to completely fill out invoices for test data.

We come away from this experiment with great appreciation of the methodological difficulties in evaluating the understanding of complex systems. First, "understanding" encompasses a wide variety of perspectives -- e.g., how the processing is done, what the input-output mapping is, what the overall purpose of the system is, the various roles played by humans in the system, etc. Second, assessment of someone's "understanding" depends critically on the method used -- e.g., recall methods may underestimate understanding while an "operating" task (having participants operate the system in some way) may over-estimate *conceptual* understanding. We feel that the latter situation may have been true in the case of this exploratory study of understanding.

Results and Discussion -- Given the above caveats, the primary finding of interest from the point of view of design is that most of the detected errors were classified as complete omissions of parts of the process. Errors of this type are especially crucial. If someone remembers that an operation is necessary, but cannot remember how to perform it, there is at least a chance that known constraints and common sense can help reconstruct the correct procedure; or, the person may know to seek outside help. However, if a person completely forgets -- or is otherwise ignorant of -- an operation, this difficulty is *not* so likely to be detected. Thus, incomplete understanding of the design problem -- on the part of either the designer or the client could lead to the production of inadequately designed systems in which the omitted processes may not be so easy to detect but may rather cause complex interactive effects (16).

5.2.3 *Cyclic Iterations in the Early Stages of Design*

In the previous work the processes and cycles of problem formulation and design were internal states of the participants, inferrable only from their behavior as tested in experimental tasks. The present study was intended to objectify and externalize these complex activities to permit us a better understanding of the initial phases of design. Rather than requiring

participants to comment on their own internal mental states (with uncertain validity), we chose to create a highly-realistic design session involving a client and a designer, whose communications, hopefully, would reveal the nature and structure of these initial processes. We also conjectured (and later investigated; see Section 5.2.4.2) that the interactions occurring between client and designer in these early design stages might parallel the mental activities of a designer working alone. If so, the objective two-person interaction might provide clues as to the nature of, and means of assisting, the single-person process.

Two staff members volunteered to assist us, permitting our video-taping of their unrehearsed interaction. The "client" was a head librarian who had a real problem of modifying some computer terminal and output equipment used for information-retrieval purposes by both the library staff and scientific researchers. The "designer" had extensive experience as a systems engineer in configuring computer software and hardware systems to satisfy such types of customer requirements.

The interaction occurred and was video-taped naturally, without special instructions, in a large experimental room with the participants seated at a table facing each other and having available large scratchpads for notes and sketches. Although no time limit was set, the design interaction was completed in 35 minutes. The audio soundtrack from the video-recording was manually transcribed, and these transcripts provided the basis for the analyses (the nonverbal visual information -- e.g., sketches, gestures, posture, facial expressions, etc. -- was not useful for the "decoding" of the design interactions; only elapsed-time measurements of the dialogue interchanges -- also obtainable from the audio record -- would be necessary to supplement the transcripts to provide a near-complete picture of the exchange).

The most simplistic idealized conception of client-designer interaction is that the former possesses only design goals and the latter only solution components, with two types of interactions: (1) successful communication of the goals from client to designer, and (2) communication of successive solution proposals from designer to client until the latter is satisfied. Given the complexity of the problem to be solved, our concern was that this model of interaction -- or indeed any more complicated model -- would be "swamped" by the complexities of verbal communication and the inherent limitations of the observational method. As it turned out, we were fortunate in recording a session which not only was extremely informative but also illustrated almost *all* of the subtleties of real-world problem-solving.

Results and Discussion -- Perhaps the most impressive finding was the sudden transformation, late in the dialogue, of the design problem from one formulation to an entirely different one. The participants had been discussing, serially and in detail, a list of the client's problem requirements, during which time the designer proffered outlines of possible solutions for each. The discussion was focussing on a very specific detail of physical equipment layout when, almost explosively, the *client* picked up on a prior notion of eliminating a piece of equipment and rapidly developed a broad generalization of the concept (see Cycle 7, p14, Appendix 1, in reference 21). The client essentially understood how the network of existing computer terminals of the user population could replace the local library equipment, whose arrangement and selection was the beginning concern of the dialogue. In the much more rapid interchange which followed, this concept was developed and proved to solve not only all the problems the client had begun the discussion with, but also provided nice solutions to ones not even considered.

This dramatic turnabout illustrates the shifting intricacy of the problem formulation aspects of design; it also highlights the extraordinary importance of permitting sufficient development of the design requirements before undertaking construction of a detailed design solution. Had the client been restricted merely to enumerating the list of specific equipment problems, with the designer restricted to verifying understanding of these, the ultimate result

would undoubtedly have been a proposal for a new configuration of local equipment -- satisfying in the short-run perhaps, but ultimately *not* the elegant solution which occurred.

Further detailed analyses of the dialogue revealed that it consisted of a series of "cycles", each cycle consisting of a regular succession of "states"-- a dialogue portion oriented towards a single purpose. The six states identified (with client and designer indicated by *C* and *D*, respectively) were:

(1) *goal statement* -- statement by C of one or more design goals

(2) *goal elaboration* -- providing more detail, especially of the context, usually by C, but often by D or in response to D's questions

(3) *(sub) solution outline* -- D's (and sometimes C's) brief suggestion of a sub-solution, usually in outline at an abstract level

(4) *(sub) solution elaboration* -- D's development of the details of the suggested sub-solution, examining its properties and consequences (often C joins in)

(5) *(sub) solution explication* -- extending consideration to other goals and solutions and examining their interaction with the present solution, by both D and C

(6) *agreement on (sub) solution* -- agreement by C to a particular (sub) solution, affirming that it is acceptable within the larger context of design goals

It should be noted that, in terms of the four-stage model of design given in Section 5.2.1, the above six states cover primarily the two initial stages of design -- problem formulation and functional requirements specification; at most state 6 extends only partially into the third stage of overall design. Thus, our observed interactions stopped far short of involving major efforts on the part of the designer.

Our analysis of the dialogue (see pp 5-13, reference 22) indicated that these states occurred exactly as ordered above, with state 1 marking the beginning of a new cycle. The complete dialogue was comprised of seven cycles. Of these, three terminated at state 4, one terminated at state 5, and the remaining three went through completion of state 6. Only one of the ordered states -- the state 2 goal elaboration -- was ever omitted within a cycle, in four of the cycles, but this did not predict the terminal state of the cycle.

The most complex aspect of the design cycles was the transition from the end of one cycle to the beginning of a new one. Whereas the overt dialogue provided clues as to the mental reasoning underlying transitions *within* a cycle, say from goal-statement (state-1) to goal-elaboration (state-2), the transitions *between* cycles were abrupt, with little overt verbal material to indicate the bases for the transition. Yet this transition would seem to be one of the most complex, since the announcement of a new goal must surely be based on the following kinds of judgments: (1) evaluation of the "value" of the material developed to that point, (2) review of the state of suspended or pending sub-goals, (3) evaluation of the overall "success" achieved so far, (4) decision to terminate the current line of development, (5) generation of alternative continuations, and (6) selection of the most promising next goal topic. Despite the apparent complexity of cycle transitions, there seemed to be little hesitation, an observation which suggests that the above kinds of processes were occurring continuously -- but unobservably -- during all dialogue states.

A second extremely complex and subtle aspect of the dialogues was the role played by the designer. In terms of verbal output, the client produced by far the greater amount, a fact

which might suggest that the designer was doing very little of the design work. It was certainly evident that the client was an articulate and accomplished problem-solver. Nevertheless, an analysis of the designer's contributions indicated that two critical leadership roles were played. The first role was that of providing the client *facts* about the "real" world: what the properties were of existing equipment, what was technically possible, alternative possibilities, etc. This contribution had two important consequences: (1) providing the client information to permit continuation of the ongoing line of thinking, and (2) permitting the client to "prune" obviously unproductive possibilities under consideration. The manner in which such information was given to the client was often by way of simple agreement (e.g., "Right", "uh-huh") to the client's assertion about a characteristic of some device; in other instances extensive detail was provided. The second directive role of the designer was in the form of *questions*, seemingly often of clarification, but frequently having the effect on the client of identifying a new problem or achieving a better conceptualization of the present problem. While we have no other evidence than the present dialogue, the flow of discussion was so smooth and productive that we suspect the designer had skillfully accomodated to the style and capability of the client so as to achieve optimal efficiency. Presumably, had the client been less articulate or knowledgable, the designer would have assumed more of the responsibility for determining the problem requirements and profferring detailed solution possibilities.

Given all of these complexities, it is remarkable that the dialogue was so structured and cyclical in nature. Certainly these regular characteristics were not evident from surface examination of the transcript or actual witnessing of the interaction; indeed, the dramatic emergence and coalescence of the all-encompassing solution in the last cycle gave the strong impression of anything *but* a cohesive structure in the dialogue. To the extent that other professional client-designer dialogues are similarly structured, there is the possibility of developing at least a *methodology* for aiding the very important early stages of design. Development of sufficiently broad and precise design requirements -- before "hardening" into specific design-solution approaches -- could greatly reduce the rework and scrapping costs which presently constitute a major component of the overall design expense.

While we have not yet verified the structured nature of our client-designer dialogue with other observational studies, we did at least find another dialogue which displayed the same cyclic structure. After failing to discover any "real" design dialogues, we did at least discover a *fictional* source containing a client-designer dialogue as we define it (in a popular adventure novel, found by the first author, 22). This work contains a conversation between the protagonist who needed a rifle with special features, and an expert gunsmith skilled in producing such customized equipment. Our analysis showed the presence of the same six states as in our dialogue, with the same characteristics of transition, showing an overall design phase of five cycles (21,22).

5.2.4 *Characteristics of the later stages of designs* (from functional requirements to high-level designs):

Whereas the above-described work was concerned primarily with the early stages of design, the rest of our research into the design process focussed on the later stages. In this work we initiated the design process at the stage-3 point of producing a high-level design (see Section 5.2.1) by providing them initially with a more or less detailed set of functional requirements for the design.

The four experimental topics discussed below are: (1) Structural design, (2) Internal iterative design cycles (3) Effects of isomorphic-presentation and representation variables, and (4) Software design.

5.2.4.1 *Structural design* -- We began our research on the latter stages of design with an investigation of *structural* design to provide a basis for determining whether this type of design differs from *procedural* design, as in the generation of computer programs.

The problem given participants was to produce detailed designs for conversion of a former church into a restaurant. They were given extensive background data on the expected clientele, the cost and spatial requirements for interior equipment, the floor plan of the church and site information, equipment and remodelling costs, and typical customer complaints about similar restaurants.

In many ways the design specifications paralleled the requirements given for software systems -- a potpourri of existing structural constraints, costs, performance expectations, etc. In particular, we were interested in observing the effects on the design solutions of gradations in the ease with which contextual constraints could be modified. In the present problem this graded constraint was in the form of the modifiability of existing structures, ranging from the least modifiable aspects of site characteristics and exterior walls of the building to the internal structural walls, to the moveable partitions shown in the church drawing.

The participants were instructed to provide a final design indicating how the interior space was to be re-configured, taking into account all of the requirement information given them. They were free, however, to express their designs in any representation -- sketches, written descriptions, etc.

Results and Discussion -- The primary difficulty encountered in evaluating the design solutions is the same as that for evaluating software designs: the product is essentially *conceptual* in nature -- detailed and elaborated, but nonetheless an abstraction; it cannot be directly tested, operated, executed, or otherwise functionally assessed. The newer methods used for checking software designs -- e.g., "peer code-checking", "structured walk-throughs" -- rely on the hope that detailed explanations by the designer to an audience of similar professionals will uncover inconsistencies and flaws in the design concept. These methods could have been used as well for the present structural design problem, and, indeed, it is our understanding from discussion with some interior and architectural designers that these methods are very frequently employed. Nevertheless, these are intuitive subjective judgments, and we sought to develop more objective -- and general -- measures of design solutions.

One measure which we applied provided a quantitative assessment of the extent to which the designs satisfied *all* of the possible sub-goals of the design problem. A thorough analysis of the goals for the restaurant layout was made on an *a priori* basis, and higher-level goals were broken down into successively more measurable and specific goals; this resulted in a goal hierarchy of three major goals --satisfying customers, employees, and owners -- each with two or three subgoals, finally terminating in three to fifteen specific and more measurable detailed goals for each subgoal. Every design was examined for the presence of the specific goal-features, and a ratio of goals present-in-the-design to total possible goals was computed. This resulted in an overall assessment of the extent to which the designs satisfied the functional requirements, a measure which we termed *Practicality*. Since our instructions did not differentially weight these detailed goals, each contributed equally to the measure. However, goal-weightings could easily be specified in the initial statement of functional requirements to provide a more sensitive measure of the extent to which the design fulfilled its

specifications. We believe this measure could be applied equally as well to other kinds of designing, including software design.

Without such a quantitative measure we would have had great difficulty in evaluating the designs or making comparisons among them, as they varied in the extreme in clarity, detail, type of representation, etc. With the measure, however, we determined that the degree of goal-satisfaction ranged from .34 to .71, with a mean of .55 Thus, the designs failed to meet almost half of the desirable goal objectives of the problem. Although we did not quantify the observation, we noted that the participants differed in the extent to which they provided detail about each of the specific goal objectives in their final design -- presumably reflecting subjectively different weightings of importance of these goal specifics. Thus, participants obviously made tradeoffs by emphasizing certain goal aspects at the expense of other aspects.

To provide a measure of the extent to which participants shared a common emphasis on specific design characteristics -- independent of whether these satisfied goal objectives -- a second quantitative measure was derived (also having potential for general use). A feature analysis was performed on all of the restaurant designs to generate a set of all the features generated by our participants. The derived measure essentially characterized the amount of feature-information present in each design relative to the total amount of feature-information. The obtained information-scores for the designs had a very narrow range in contrast to the goal-satisfaction measure, from .63 to .77. While the expected value of this statistic is not known these relatively high values indicate that the designs tended *not* to emphasize the same set of features; rather, the different designs appear to have focussed on different feature subsets, perhaps due to different interpretations of the requirements or even personal preferences or competencies. There also was the suggestion of an inverse relationship between this measure, which can be interpreted to assess something like design "originality" and the above goal-achievement or "practicality" measure, such that several designs were found to be high in originality but low in practicality, and vice versa.

The above two findings suggest that methods for structuring the design process could be useful in (1) achieving completeness with respect to design goals, and (2) controlling the emphasis placed on design features.

With respect to the effect on design of the graded modifiability of the initially-given building structures (e.g., site, walls, partitions), we found that participants' designs accurately reflected this factor. For example, only three participants changed the external walls, 12 changed the internal structural walls, 16 changes the non-structural walls, 18 changed the moveable partitions, and all 27 participants changed the furniture or architectural detail. While such an accomodation to the given difficulty of structural changes is understandable (and perhaps justifiable on cost or other grounds), such pliant responses also pose a danger for the ultimate quality of some kinds of design: in many cases the most successful designs are characterized by a boldness in rejecting existing structures (even if extensive modification is required) to achieve a more coherent and functioning integrity. The obvious structural example is that of the current practice of modernizing an older structure by tearing down whole walls, structural or not, punching windows into solid wall expanses, etc. But the same can be true for software designs, such as when a new data-processing system may achieve success partially by its complete reconstruction of all input and output data structures.

In addition to the above results we also obtained information concerning participants' "styles" in designing -- the decision strategy controlling their sequential development of the design. Because of the graphical nature of this design problem it was not feasible to attempt to record sequential activities of participants; rather, we solicited information concerning their design strategy via post-design questionnaires. Two-thirds of the participants reported the following style characteristics. They claimed to have: (1) used a top-down approach,

employing successive decomposition of larger problems into sub-problems, (2) planned out their approach before beginning, and (3) worked on the hard parts first. It is of interest to note that, with the possible exception of the third characteristic, these participants' design styles in the design of a structure are in accordance with the consensus recommendation for designing software procedures.

In summary, the results of this experiment produced a detailed picture of the process of *structural design* which seems remarkably in accord with what we believe is true of *software design* (17).

5.2.4.2 *Internal iterative design cycles* -- In the two-person client-designer experiment (Section 5.2.3) we detected a high degree of structure in the dialogue, represented by regular cycles of transitions between problem states. In the present study we sought to determine whether similar such cycles occurred during the development of a design by a *single* person. The "Client" role was simulated initially by providing participants with an unorganized but detailed list of requirements for a design problem. Thereafter, of course, the participant received no further external communication.

In the experiment the participants were instructed to design an organization of a hypothetical library, given a list of 22 specific library procedures (e.g., "books left out in the reading room must be reshelved"), and knowing that a staff of 10 librarians were available. They were asked to write down comments about what they were doing when, how they were doing it, and why. This information provided the basis for assessing the presence of cycles in the design. Analyses of the final design solution provided information about the manner and commonality of the classification organization imposed on the tasks. This information also permitted corroboration of proposed design cycles.

Results and Discussion -- The major finding was that there was indeed a strongly-structured cyclical nature to the design process. And, like the previous client-designer dialogue, similar states composing the cycles were found, with similar transitions to a new goal-cycle made before completion of the cycle to full sub-goal acceptance. The two situations differed, however, in that the present intra-cycle states were more similar to the classical problem-solving scenario of successive decomposition of goals into subgoals, following a goal-hierarchy. In the client-designer dialogue there was much more of a *communicative* quality of exposition and elaboration of the speaker's meaning. Presumably, this larger communication component in the dialogue was due to the exigencies of inter-personal meaning transfer (see 15), whereas in the present experiment the participants would, of course, have no such problems in communicating with themselves.

The goal-decomposition nature of the design process reported in the present experiment (and also in the experiment on structural design, see Section 5.2.4.1) was *not* like the typical artificial-intelligence (AI) system's depth-first descent into the goal tree, with minimal backup to next-higher goals in the case of blocking. Rather, the present performance was different in two respects from such algorithmic AI processes. First, in most AI situations the goal-hierarchy is relatively stable, based upon the initial problem formulation, with new goals added primarily only as decompositions of higher goals. In the present case the participants' reports indicated *dynamic* restructuring of the goal-hierarchy -- e.g., as unforseen difficulties arose or inadequate conceptualizations were detected. Second, while the overall tendency was that of local depth-first decomposition, there were frequent instances in which participants abruptly terminated effort towards solving a ("low-level") subgoal and began work on a much different (and "higher") goal (in the inferred goal-hierarchy). (20; also 21)

Assuming that these processes also characterize software design, these observations have an important implication concerning aids for supporting the programming process. Consider the proposition that software development *should* follow a top-down successive-decomposition approach. There is considerable consensus among software developers on this point, and some of our experimental data suggests that this may even be a natural, as well as a desirable, method of design. One very effective means of enforcing such an approach is to have designers evolve their program designs within a computer environment which *requires* all development work to be either a decomposition of a larger unit or an extension of a unit. There are, in fact, a few such systems of this type in limited use at the present time. Our present data suggest, however, that these systems may be much too inflexible to support the dynamic shifting of goals and goal-structured activities which appear to characterize the natural design process.

5.2.4.3 *Effects of Isomorphic and Representation Variables* -- A key distinction between structural design, such as in architecture, and the design of procedures is that the primary relationship between design elements is *spatial* in the first type of design and *temporal* in the second. Our intuitions, as well as various kinds of suggestions in the literature, led us to hypothesize that temporal designing might be more difficult than spatial designing, because of greater experience and heuristic-availability for the latter. In the first of two experiments reported on in this section we compared the two types of design and found temporal designing to be more difficult. In the second experiment we provided a structured representation for encoding the problem information and observed the diminishment of this effect.

Experiment 1: Temporal vs. Spatial Designing. -- In comparing the two types of design it was necessary to maintain constant the conceptual difficulty of the problems in the two modes. This was accomplished by selection of a problem which had two isomorphic versions, differing only in the nature of the key content words, these being "temporal" in nature in one case and "spatial" in the other. The cover story for the temporal isomorph involved designing a manufacturing process for "widgets" consisting of seven stages. Various types of information about the stages were supplied, such as priority, sequence, and resource-utilization relationships between stages. For example, three pieces of information given in the temporal isomorph problem were: (1) Stage F has a higher priority than Stage B, (2) Stage A should follow Stage C, and (3) Stage G uses different resources than Stage F. Each stage was to be assigned to a factory work-shift such that stages using the same resources should be assigned to the same shift, and the total number of sequential shifts should be minimized with high priority work accomplished first.

There were a total of 19 different functional requirements specified for the temporal problem, and each had an isomorphic form for the spatial problem. The "cover story" for the spatial problem was the design of a business office layout which was to accomodate seven employees (corresponding to the seven stages) such that the total number of corriders required were minimized, subject to the interrelationship factors between employees. These factors were "compatibility" or "incompabitiliby" (whether employees get along, corresponding to using the same resources or not, in the temporal problem), more or less "prestige" (corresponding to priority), and need for proximity to one or another part of a main corridor (corresponding to sequencing). Three example functional requirements for the spatial isomorph, corresponding to the examples given above for the temporal version are: (1) Person F has more prestige than person B; (2) A meets people in the reception area more than does C; and (3) G is incompatible with F.

The 19 functional requirements were given to participants under various types of grouping relationships, but this variable had very little effect in either this or the next experiment. In

this first experiment, participants were asked to provide a partial "design" based on the functional requirements they had received up to that point; they were also asked to give a final design based on all the functional requirements at the end of the experiment. No suggestions were given either concerning how to represent the requirements information or to structure their partial and final "designs." The wording of the requirments and the overall conceptual structure of the design problems were equivalent for both types of problems; any performance difference could be attributable only to pre-experimental baises for or against the temporal or the spatial relations and operators. Seventeen participants were used for each isomorph group.

A reliable performance difference was found between the spatial and temporal groups, with the temporal isomorph solved more slowly and less successfully. Examination of the representations participants used to portray their design solutions revealed a remarkable difference: all 17 participants in the spatial isomorph condition used a graphic representation of the business office to portray their design solution (a rectangular top-view drawing). However, only two of the participants in the temporal isomorph used such a representation -- even though a graph is ideal for portraying both types of problems. This latter findings implies that the differences in performance with the isomorphs might be attributable to the availability of representations under the two conditions, and not to more fundamental problems of greater difficulty with temporal vs. spatial concepts per se.

Experiment II: Effects of making available a design representation. -- This experiment was identical to the first, with the exception that all participants were supplied with blank graph segments (roughly 8 x 8 cells in extent) within which they were to record their partial and completed designs. Twenty-two subjects served in each isomorph problem condition.

As a result of providing participants with the graph for representing their design solutions -- particular the temporal isomorph group -- the differences in performance between the two conditions were drastically reduced. Thus, it does appear that, even though the two problems are conceptually isomorphic, the expression of the problem in terms of temporal concepts produced greater unavailability of effective means of representing the requirements and design information.

Despite the fact that providing a representation aid reduced the performance differences, there were still significant differences between the two isomorph conditions in terms of the number of participants who failed to comprehend the problem; further, this difference in comprehension was the same in both experiments, and was therefore unaffected by provision of the graphic aid. Assuming, for discussion, a simple two-stage model of problem-solving with *problem-understanding* followed by *problem-solution*, the findings of the two experiments can be interpreted as follows: (1) temporal design problems are more difficult than spatial design problems in both problem-understanding and problem-solution; (2) problem-solution difficulty for temporal design problems is partially produced by the unavailability of effective information representations; (3) making available powerful information representations for temporal design problems improves the problem-solution stage; (4) problem-understanding is more difficult for temporal than for spatial design problems, and this apparently is not reducible by representation aids. (18).

We belive these experiments illustrate the importance of providing support tools -- including representation aids -- for the design of programs. This task appears to be inherently more difficult than other kinds of design tasks, and therefore should receive more assistance.

5.2.4.4 *Software Design* Our last research project concerning design involved the actual design of a software system by experienced programmers. We asked eight programmers with

four to 12 years programming experience to take a set of functional requirements for a relatively limited query system and produce a detailed design for the software required. The design was to be expressed in natural language, but with sufficient precision that it could be translated fairly directly into (PL/1) code.

The specifications for the problem were very carefully worked out and provided details for all aspects of the query system to be produced, including: (1) input syntax, (2) requirements for internal file manipulations, and (3) output syntax. Performance efficiency (e.g., speed of execution, amount of computer memory required) was de-emphasized, and "usability" and clear design structure were stressed. In terms of the design model given earlier (see Section 5.2.1), this experiment began at the end of Stage 2 -- with a very clear statement of functional requirements -- and continued through Stage 4 -- to the level of a detailed design.

Results and Discussion -- The most striking feature about the design solutions was their diversity. They varied considerably in terms of their production characteristics, their form -- or syntax -- and their content. In terms of production, 3 to 6 hours were required to complete the designs, and they varied in length from 95 to 199 sentences (706 to 2345 words). In form, they went from a highly abbreviated linguistic style very reminiscent of programming, using indentations and several other features of programming languages, to a very discursive "long-winded" narrative style. It was in content, however, that the greatest diversity was observed. With respect to the overall query-system algorithm, three programmers did not supply specific details while the remaining five described five different solutions to the problem. Each programmer tended to focus on a different part of the problem, providing greatest detail there and correspondingly less detail elsewhere; for example, some participants worried about the user's input of queries (but again concentrating on different details), while others were concerned with internal file manipulations, and still other concentrated on the output features. This rather extreme degree of diversity is all the more surprising in view of the fact that all participants were given the same instructions and the same set of very detailed functional requirements. Furthermore, the programmers were very homogeneous with respect to programming experience and other potentially relevant variables, so these individual differences do not seem to be a likely cause of the diversity. We conjecture that it was the programmers differential experience and competency with different aspects of the problem which contributed most to the diversity of focus and content. These observations implie the fundamental intrinsic difficulty of designing programs -- as evidenced by the present diversity --and confirm such suggestions determined from our other studies.

The second major aspect about this study was the presence of two different "styles" of designing, which seem to be related to the quality of the design solutions. The first, "programming", style (PS) is characterized by a highly formatted text, use of programming control commands (e.g., "GOTO"), with terse, objective, and impersonal writing. The second, "narrative", style (NS) has no features characteristic of programming, but is similar rather to extended narrative writing -- with paragraph structuring, use of personal references, and replete with extended descriptions and subjective evaluations. Based on these very general characteristics, we found that an equal number of programmers fell into each group. We then performed more quantitative analyses on the two groups, first in terms of sentences and overall words, and then in terms of the types of words used. With respect to the overall measures, although PS required 30 percent more time than NS, they were much briefer in their expression of design solutions; NS required: 25 percent more sentences, 61 percent more words, 68 percent larger vocabulary, and 97 percent more words per sentence. As for the types of words used in the two groups, we made several observations on the apparent part-of-speech classes of the 150 most frequently used words of each group. Several differences appeared: PS used fewer adverbs, NS used a much wider variety of verb forms (e.g., present and past participles, more frequent use of modals like "should"), and NS used

roughly three times as many verbal quantifiers (e.g., "each, all"), deictics (e.g., "this, that"), and personal pronouns.

The last major observation from this experiment -- perhaps the most provocative but also the least quantified -- is that the major characteristics of design *quality* appear to be predictable, to some extent, from the overt characteristics of *style*. More specifically, we observed that the PS design solutions tended to be more complete, have fewer obvious errors, use more inefficient algorithms, etc., than the NS solutions. It appears, therefore, that there may be some hope in predicting design quality from overt style characteristics without too much further analysis of content.

- 5.3 *RECOMMENDATIONS:*

Our investigations obviously only just begin to provide information about the very complex task of designing. Much much more behavioral research is required before the task is sufficiently well-known to permit development of highly suitable and tailored tools and support environments. Still, we do have a number of recommmendations which we feel are sufficiently sound as to warrant their investigation at the present time. of designing presented earlier.

Our recommendations are therefore presented in two sections, one dealing with the additional types of experiments we believe would be most informative, and the other discussing specific suggestions for design tools or aids.

5.3.1 *Behavioral Experiments*

5.3.1.1. *Design philosophies* -- The most pressing behavioral problem seems to us to be that of obtaining some comparative quantitative data on the relative characteristics of the various design philosophies. One or the other of these philosophies are being adopted, *by fiat*, by programming organizations, with very little evidence supporting their effectiveness -- other than crude case-study reports or anecdotal observations. While it may be true that performance gains can be achieved just by the simple fact of all of the programmers using the same techniques, still the costs of software design -- and errors in them -- are so high, that some empirical investigations are amply justifiable.

We recommend that a behavioral methodology (e.g., task, subjects, programs, and problems) be established to permit comparative evaluations of at least the following techniques (the person most associated with the technique is given in parentheses, along with their affiliation): Top-down structured programming (Mills, IBM), Modular programming (Myers, IBM), Structured design (Yourdon of Yourdon, Inc.), and Program Specification Language (Teichroew, Univ.Michigan ISDOS project).

5.3.1.2. *Linguistic style of expression* -- The results of our last experiment, studying software design, suggests the intriguing possibility that significant gains in design quality may be achieved, not by following some complex design philosophy, but rather by a much simpler adherence to overt stylistic conventions in expressing the design. As an interum step in improving software design, we therefore recommend that an experiment be conducted to assess whether designs produced according to the following types of stylistic conventions result in productivity or quality gains. Such conventions would include: (1) use of only imperative sentences (as opposed to declarative ones), (2) avoidance of embedded phrases or clauses, (3)

emphasis on providing highly-formatted segregation of different content statements (by means of indentation, use of labels, etc.), and (4) minimizing the use of qualifiers within sentences, such as adjectives or adverbs, by means of using a greater number of individual separate sentences to express these qualifications.

5.3.1.3. *Use of semantic data-types* -- Throughout this report we have suggested the possible utility of testing the internal consistency of high-level designs by means of characterizations in terms of data-types and semantic restrictions between operators and defined data-types. This approach requires the designer to evolve a set of descriptive data-types to be assigned to each data variable, with each operation on variables to be characterized in terms of which data-types are permissable inputs to the operators and what the resulting data-types are after operator application (see Sections 2.3.2, 3.3, and 4.2.1.2). We recommend that this approach be implemented for testing under laboratory conditions to permit evaluation of the potential utility of this approach.

5.3.2 *Specific software design aids*

5.3.2.1. *Checklists* -- There are a number of steps in the design process where the use of pre-established checklists might be useful in improving the completeness and the organizational coherency of the design solutions. In the very first step in which the client communicates the problem requirements to the designer, two types of checklists could be generated to assist this phase: (1) a checklist could be provided the client *before* the problem is presented to the designer, to insure that the client had considered ahead of time all of the important aspects of problem specification, including statement of goals, relationships to be effected between other systems or software, time/money/manpower resource constraints, etc; (2) a similar checklist could be provided for the client-designer interaction to insure that all aspects of problem definition were sufficiently covered in their discussion. In the stage for developing a high-level design from functional requirements, a structured checklist could also be developed to insure that the designer had attended to and successfully represented solutions for all of the separate design requirements. Finally, a checklist could be provided to insure that all the necessary detail was provided in amplifying the high-level design into the very detailed level of design from which to generate the actual program code (e.g., specification of input and output interfaces between adjacent procedures, checking conformity of design aspects with requirements for linear control transfer, for passing of variables into or out of procedures, for appropriate operation sequencing -- these based on the best "conventional wisdom" available).

We recommend that these checklists be developed as aids for the design process and then suitably evaluated in laboratory performance tests.

5.3.2.2. *Catalogues of process algorithms* -- We believe that the complexity of software design can be reduced very significantly by permitting designers to *select* appropriate sub-processes rather than to *generate* them in the design (selection being well-known to be a simpler and less error-prone behavioral task than generation). For example, consider the situation in which a designer is elaborating how some large data file is to be searched for some target data. In our last software design experiment we observed that several of the programmers devoted a relatively large proportion of their total design content to specifying file-search processes, with a great deal of diversity in the method described. As an alternative, we recommend that catalogues of various kinds of processes be made available -- either on- or off-line -- such that the designers could specify their requirements for a process and then determine what techniques were *already designed and available* to accomplish the desired purposes. Thus, for specifying a file-search process, the designer would access the catalogue with a description of the process sought -- e.g., a file-search --along with the particular

constraints on the process -- e.g., the organization of the file (such as "indexed-sequential"), whether the data sought is a "key" field in the file or an embedded field, etc. A pre-established process would then be identified in the catalogue, giving the name of the process, the input/output characteristics, etc. The designer would then simply indicate in the design that a prior-defined process was to be inovked at that point, giving its name and reference information. We believe that the detailed aspects of software systems often require use of a large number of processes which could be standardized across applications, written for generalized use, and catalogued for relatively easy *selection* by means of a catalogue-retrieval information system. Every time a designer selected such a standardized process, as opposed to writing a similar one "from scratch", many careless as well as conceptual errors could be avoided.

We therefore recommend that the utility of this approach be tested by accomplishing the following steps: (1) categorize software applications by type so as to maximize the number of commonly used processes across all of the applications of one type (e.g., one type of application concerns *information systems*, another is *real-time interrupt processing* each application involves a number of processes common to each type, but different between types); (2) identify the high frequency detailed processes involved in each application type, and write generalized programs to accomodate all of the variations; (3) develop a highly-indexed catalogue of the pre-established processes for each application type, such that designers can access the information via a variety of synonyms and descriptors; and (4) make these catalogues available for designers during their actual design development activities.

5.3.2.3. *Hierarchical program editor* -- We believe there is sufficient evidence, from our observations as well as from case-studies, to believe that the "top-down" or successive decomposition approach is compatible with the intrinsic behavioral nature of software design and also offers significant advantages for improving the productivity of designers and the quality of their designs. We therefore recommend the development of a specialized program/design computer editor to facilitate designing according to this philosophy.

The central concept underlying this method is that processes are organized *hierarchically*, with the degree of detail concerning the processes being an increasing function of the "depth" within the hierarchy. If designers are to create software systems based on hierarchical organization, they should be aided in this process by specialized computer editors which facilitate all of the hierarchical manipulations that will be required -- e.g., searching for targets at various hierarchical depths, inserting or deleting within hierarchical stages, copying or transferring parts of the hierarchy from one place to another, etc. We know of no editor -- even as a research project -- which could adequately support such activities, providing all of the desirable unburdening features requisite to full support of this approach. Present program editors -- even the most advanced full-page editors -- provide for organization of information at only one level.

Accordingly, we recommend that a hierarchical editor be developed (and evaluated) to support this design philosophy. We use the following hypothetical example of an information system being developed hierarchically, in which each numbered statement is the descriptive identifier of a process eventually to be elaborated into programming code; the example is incomplete and represents an early stage of specifying the design, with greater detail given some aspects than others.

```
1. OBTAIN NEXT INFORMATION QUERY
  1.1 CHECK WORDS IN THE QUERY AGAINST DICTIONARY
     1.1.1 TOKENIZE WORDS IN QUERY
     1.1.2 DO UNTIL END OF FILE OR UNTIL "ERROR"
        1.1.2.1 GET NEXT WORD
```

```
   --->1.1.2.2 SEARCH DICTIONARY FOR WORD
       1.1.2.3 RETURN "ERROR" IF NOT FOUND
  1.2 CHECK QUERY SYNTAX
    1.2.1 SEGMENT QUERY INTO TRIPLES OF
          ATTRIBUTE-OPERATOR-VALUE
          (EX: "AGE NOT-GREATER-THAN 30")
2. SEARCH FILES FOR DATA SATISFYING QUERY
3. OUTPUT RESULTS OF FILE SEARCH
  3.1 DETERMINE IF ANY MATCH FOUND
    3.1.1 IF NO MATCH, ISSUE APPROPRIATE MESSAGE
  3.2 DETERMINE APPROPRIATE OUTPUT FORMAT
    3.2.1 CHECK USER'S PROFILE FOR FORMAT DETAILS
    3.2.2 IF NO FORMAT GIVEN, ISSUE REQUEST
          TO USER FOR FORMAT DESIRED
  3.3 PREPARE OUTPUT ACCORDING TO SPECIFICATIONS
  3.4 WRITE OUTPUT ONTO DISPLAY DEVICE
```

We now describe some of the functions which a hierarchical editor should perform for the designer. For purposes of illustrating these functions, assume that the user has just completed entering the line numbered 1.1.1.2, as indicated by the arrow in the above figure. Because we believe that a hierarchical editor offer the greatest potential for facilitating the design process, without awaiting further research efforts, we describe the functions in extensive detail, to indicate the complexity required for adequate support.

1. *Controlled display* The user should be able to selectively display various aspects of the system under development. The parameters for controlling the display should include the following: (1) the *reference points* for the display, at least three such --(a) from the "top" and beginning of the design, (b) from the point of the last entry or modification (or search) in the design, and (c) relative to any arbitrarily specified level; (2) the *direction* for the display, again three such -- "up" to a higher hierarchical level, "down" to a lower level, or "across" to entries at the same hierarchical level as the reference point (the latter has two further options -- "forwards" and "backwards"); (3) the *extent* of the display --the number of levels to be shown, specifiable as either 1, "all", or some arbitrary number. Assume that the syntax of the display command is: "SHOW (REF.POINT) (UP/DOWN) (EXTENT) (ACROSS-EXTENT)", with omission of arguments defaulting to the last prior reference, for REF.POINT, and to ALL, for EXTENT parameters. The command "SHOW UP 2" would then result in display of all hierarchical levels from 1.1 down to 1.1.2.2; similarly, SHOW TOP would result in display of the whole hierarchy.

2. *Addition of new material* -- The designer should be permitted to enter new material anywhere in the hierarchy without having to traverse (up or down) that hierarchy. Thus, the editor should support a request to enter new material with a parameter indicating the location -- e.g., "INPUT X", where X is the location identifier. Assume the designer wishes to enter the line "ERROR CHECK". Given the last entry at 1.1.2.2, if this new line were simply entered without a location specified, the editor would assume that it was to be inserted at the same hierarchical level as 1.1.2.2 and would assign it the line number 1.1.2.3, renumbering any lines following (thus the existing line 1.1.2.3 would be renumbered as 1.1.2.4).

Location specifications would be parti[illegible] The *partial location* entry "INPUT .10 ERROR CHECK" would thus resu[illegible]OR CHECK" being inserted as the lower-level line "1.1.2.2.10 ERROR CHECK", a subsequent entry of ".7 EOF CHECK" would result in an insertion of the line "1.1.2.2.10.7 EOF CHECK" (it not being necessary to repeat the INPUT token again, once the input mode is established). A *complete location* entry such as "3.1 EOF CHECK" would result in the entry of a new line "3.2 EOF CHECK", with

renumbering of the line 3.3 to 3.4, etc. Finally, designers should be permitted to add new material which is not yet to be integrated into the existing main design hierarchy -- to permit them to develop portions of the program when they are best able to do so without having to specify at that time where these portions are to fit in. Thus, a prefix indicator such as "U" could indicate that the new material was to be *unattached* to the main design tree, such that the entry "U.2 1.2 EOF CHECK" would result in the creation of a separate hierarchical tree with the numerical identification of "U.2" and an entry of "1.2 EOF CHECK" into this hierarchy; a subsequent entry of "U.2 1.2 ERROR CHECK" would insert the line "1.3 ERROR CHECK" into the U.2 hierarchy, with appropriate renumbering.

3. *Movement of material* -- The hierarchical editor should also support facile movement of material from one part to another, there being two basic modes: (1) destructive move --material is moved to a new location and deleted from the old, and (2) non-destructive move -- material is copied from an old location to a new one, without destruction. The syntax of such a move command might be: "MOVE/COPY (OLD LOCATION) (NEW LOCATION). Thus "MOVE 3.2 to 1.2.1" would result in the relocation of the lines 3.2, 3.2.1, and 3.2.2 to be the first lower level under 1.2, the existing line 1.2.1 (and its lower-level components) being renumbered with the leading digits 1.2.2, and the lines 3.2, 3.2.1, and 3.2.2 being renumbered as 1.2.1, 1.2.1.1, and 1.2.1.2 respectively. Substitution of the word "COPY" for "MOVE" in this example would leave the 3.2.X lines as they appeared and insert a copy of them as described above. Such a facility gives the designer tremendous power in restructuring the design -- an action which occurs very frequently during software design.

The MOVE/COPY command would also be used to move (or copy) independent hierarchies (such as the "U.2" example) into the primary hierarchy: "MOVE U.2 2.1 3.4" would thus result in the destructive relocation of the partial hierarchy under the node 2.1 in hierarchy U.2 to the location 3.4 in the primary hierarchy.

4. *Controlled context search* -- The designer should be able to search the hierarchical design for occurrences of any procedure or variable -- or any classes of these --subject to specifiable conditions of (1) hierarchical level and (2) context. The syntax of this highly complex function might be: "SEARCH (FROM REF.POINT) (DOWN EXTENT) FOR (NAME) (IN CONTEXT "X")". Thus, "SEARCH FROM 1.1 DOWN 1 FOR 'DO UNTIL' " would result in line 1.1.2 being located. Additional power could be provided by permitting the designer to define *categories* of names, such as words indicating transfer-of-control (e.g., "IF, DO, CALL", etc., identified under name "CAT.CONTROL"); then the command "SEARCH ... IN CONTEXT CAT.CONTROL" would result in a search for the designated target in the context of any word defined as a transfer-of-control entity (the exact specification of what is meant by "context" could have several parameters, but this is not treated here).

5. *Documentation* -- Finally, the designer should be given the capability to attach a variety of documentation and other information to any node in the design hierarchy. The hierarchical structure represents essentially the "calling" structure among separate procedures, where, in network terms, the relation between nodes are specialized transfer-of-control relations -- e.g., there is an arc from 5.1 TO 5.1.1 labeled something like "CALL to", with a reverse arc from 5.1.1 to 5.1 labeled "CALLED FROM", with the two arcs relating 5.1 and 5.2 labeled "TRANSFER TO" and "TRANSFERRED FROM". We suggest that *additional* classes of arc-relations be defined to permit association of other kinds of information with each node. Thus, a "Document" relation may be defined which associates explanatory material with a node; similarly, "I/O" relations may be used to describe the input and output characteristics of a node. By means of these separate classes of relations any amount and kind of information can be directly associated with the primary procedure-level statements, and the display command would have an additional parameter to show or to suppress this data.

We have described in extensive detail the possible characteristics of a hierarchical editor to illustrate (1) how complex and sophisticated editors to support software design could be, and (2) the "aiding" and "unburdening" that can thereby result for the designer. We have in fact developed a prototype of such an editor -- with the above features and many more -- and very limited testing of such an editor confirms our belief in its utility. However, we lacked the programming resources to develop a high-performance version of the editor for more extensive testing.

6. *INVESTIGATION OF PROGRAM MODIFICATION BEHAVIOR*

6.1 *MOTIVATION:*

The research described in the previous sections has been concerned with the behavioral problems in *generating* computer programs. The completion of a well-designed, well-coded, debugged and tested program is not, however, the final stage of the software-development process. Following delivery of the software package to the "customer" there begins a long and costly process of providing support services for the program, known as *program maintenance*. The maintenance services can be required for many years -- as long as the program is used -- and the cost is estimated as being at least two-thirds of the overall total cost of the program product, including design and development. These services include the following: (1) correcting undetected syntactic errors -- "bugs" -- in the program; (2) changing parts of the program, which, although without "bugs", do not conform to the original design specifications of the program; (3) modifying the program in accordance with *new* design requirements -- e.g., to provide additional function. We call the last two services *program modification*.

For every aspect of the programming process prior to program modification there are various tools and techniques which can be used to facilitate those phases. Thus, there are techniques for obtaining "well-structured" designs and coded programs; there are a number of specialized debugging tools, in addition to the debugging information supplied by language compilers (and interpreters); and there are a host of specialized techniques and programs for exercising and testing the data-manipulation and control-flow characteristics of the software. In addition, the programmers who perform one aspect of the process can most often consult with those who perform the other aspects -- as the program testers can ask help from the program designers with respect to testing complicated programs. Indeed, these separate programming functions are often performed by the same group of persons, such that any phase can be facilitated by their considerable knowledge of the purpose of the program and its algorithmic characteristics.

For the program modification process, however, there are *no* specialized assistance tools. To make matters worse, the programmer who is to make the modifications typically has never seen the program before and therefore has no prior understanding of its purpose or its functioning. The documentation on the program is almost certain to be inadequate and out-of-date, and the programmers who once understood the program are most likely long gone. Finally, the description of the modification to be made will usually be sketchy and offer no clues as to how the modification is to be made and where.

Despite these difficulties and despite the obvious importance of the modification process -- both in terms of the labor-cost and the potential impact on program performance --case-studies and research investigating this aspect of programming behavior are almost entirely lacking. We therefore decided to conclude our intensive study of the various phases of programming with an exploratory analysis of and investigation into the modification process, being particularly concerned with the identification of useful tools.

6.2 *SUMMARY OF WORK:*

Our exploratory work into program modification is organized into the following six subtopics: An *a priori* model of program modification behavior (6.2.1), hypothesis of most

difficult processes (6.2.2), considerations for tool development (6.2.3), modification tools tested (6.2.4), and the behavioral findings (6.2.5).

6.2.1 *A priori model of program modification*

Modification of programs is conceptually a much different task than either the design or the coding of programs. The program modifier typically is concerned only with an isolated performance or functional aspect of the program; there is no need to be concerned with the overall design of the program, the general quality of coding, overall performance, or indeed any other aspect of the program besides the very local modification which is to be made. The situation is very much like that of an electrician or carpenter called in to modify some part of an existing dwelling: their job is *not* to rebuild from the ground up, or even to critique the design; their objective is to put in place some modification of the structure which accomplishes the desired goals without otherwise detracting from the existing qualities of the dwelling.

To provide us a better initial understanding of the modification task we informally interviewed application programmers, asking them about the typical kind of modification they made to programs. The type of modification most commonly mentioned involved modifying a *transaction* within the system, making provision for one of the following: (a) addition of a new class of information to be input to and processed by the system, (b) addition of a new class of information to be output by the system, (c) changing the manner of internal processing of a particular type of information, or (d) changing the manner of selecting and formatting particular output information. In contrast to these types of changes, few programmers mentioned modifications to the overall program structure or to the documentation. The primary motivation cited for the changes was that of *functional enhancement* as opposed to performance-efficiency or readability improvements.

We also asked programmers to describe the steps they went through in making modifications to programs they did not know, and on which they had very little documentation. Based on these observations, and the results of our own introspection in performing such program modifications, we developed a tentative *a priori* model of the conceptual processes and steps involved in program modification. Our purpose in articulating such a model was to determine which aspects of the modification process might be most difficult and error-prone; these aspects would most likely be the best candidates for providing support tools and techniques (see the following section, 6.2.2). This model consists of 8 stages and is described below.

1. *High-level understanding of the modification* -- A very sketchy idea of the nature of the change to be made is obtained. The programmer's impression of the complexity of the change may well lead to the suggestion that a new program be written from scratch. This stage, of necessity, often co-occurs with the second stage.

2. *High-level understanding of the program* -- A similarly sketchy understanding of the purpose of the program, as well as some idea of the overall logic, is also obtained. In addition to reading the documentation, the program will also most probably be examined to provide the programmer a "feel" of the program's logic and complexity.

3. *Detailed understanding of the modification* -- The programmer's understanding of the modification is elaborated, focussing on two aspects: (1) the nature of the data-transformation operations to be performed, and (2) the data structures involved in the modification -- particularly the *files* that will have to be manipulated. During this

However, goal-weightings could easily be specified in the initial statement of functional requirements to provide a more sensitive measure of the extent to which the design fulfilled its

stage, possible ways of achieving the implied operations and manipulations are considered, based upon prior experience and the nature of the program itself.

4. *Gross mapping of modification to program* -- The program is examined in somewhat greater detail to determine where the required operations might possibly be performed; similarly, the data structures implied by the modification statement are sought for in the program code. Difficulties in determining the mapping relationships at this point may well initiate a much closer examination of the program -- e.g., using compilation information, conducting a detailed trace of the program logic and data flow, or even executing the program.

5. *Generation of implementation approaches* -- Various ways of implementing the modification are considered. This consideration extends to locating the specific code which might be modified or added to, and it may include writing of trial replacement code which might be inserted. However, the program is not actually modified in this step. The following kinds of considerations are taken into account: (1) minimizing the overall amount of new code to be written, (2) minimizing the number of affected independent program components (these components are variously called "sub-routines", "modules", "external procedures", and simply "procedures"; the last is used here), (3) minimizing, in general, the possibility of introducing "side-effects" which modify function or performance of other parts of the program, (4) maximizing performance efficiency, and (5) maximizing the generality of the change, in anticipation of other similar modifications in the future.

6. *Selection of an overall modification plan* -- This additional step is appropriate when the modification is extensive and involves changes in a number of different parts of the program. The programmer may well decide to change the overall control structure in some way to better accomodate the modifications within a separate procedure. The individual parts of the modification might themselves not require such control-flow restructuring, but program efficiency may thereby be increased -- as well as decreasing the chances for undesired side-effects.

7. *Implementation of the modifications* -- Changes are made to the actual program code to implement the chosen method of modification.

8. *Modification debugging and testing* -- These operations are identical to those involved in programming a new application, except that the programmer -- hopefully -- has two versions of the program and is in a somewhat better position to determine the effects introduced by the modifications.

6.2.2 *Hypothesized Difficulty of Tracing Processes*

Stages 1, 3, 7, and 8 involve processes quite similar to normal programming (if understanding of the modification in stage 1 is equated to understanding of the initial program requirements). The key difference between initial programming and program modification is therefore believed to be the programmer's lack of understanding of the program. Thus, stages 2 and 4-6 would appear to be the most difficult stages of modification, all requiring that the program be understood in various ways. We believe this understanding involves two different aspects of programs: the *data-flow* and the *control-flow*. In the first two following sub-sections we hypothesize details of the processing difficulties expected for each of these types of flow. In the third sub-section we discuss a type of data-flow tracing which also requires knowledge of the control-flow.

design strategy via post-design questionnaires. Two-thirds of the participants reported the following style characteristics. They claimed to have: (1) used a top-down approach,

6.2.2.1 *Understanding data-flow* -- This type of understanding involves tracing of the means by which one data-variable gives its value to succeeding variables. For example, in the sequence of algebraic equations "x = 5 + b, y = 8 + x, z = 10 + y" each of the variables *x*, *y*, *z* is a result of (an addition) transformation on two input variables, one of which is a constant and therefore has no prior derivational path. The other input variable, however, derives from previous operations, such that, for the variable *z*, this second input can be traced backwards to variable *y*, thence to *x*, and, ultimately, to *b*.

We hypothesize three types of data-flow tracing within program procedures as being important in program modification: (1) *forwards input tracing*, (2) *backwards output tracing*, and (3) *internal variable tracing*. A fourth type of tracing, *between-procedure variable tracing*, is discussed separately in section 6.2.2.3. In terms of the four types of modifications most commonly cited (see the beginning of section 6.2.1), it would seem that type 1 *forwards input tracing* is most important for understanding how to incorporate a new or modified transaction input (modification type a); in addition this type of tracing would be most important for achieving an initial overall understanding of the program. Modifications of the output (type b) or modified output format (type d) would primarily -- or at least initially -- require type 2 *backwards tracing*. Since most programs typically involve more inputs to a variable than outputs from it, backwards tracing is assumed to be a more complex process than forwards tracing, involving -- for the same number of steps -- consideration of more variables and more flow paths. Type 3 *internal variable tracing* involves tracing both forwards and backwards from some internal variable in the program (i.e., the variable is neither an input nor an output); we presume that this type of tracing is most closely associated with the modifications of internal program processing (type c). In internal tracing, the programmer locates the occurrence of a data variable which is of interest and follows the data-flow paths into and out of that variable for some number of steps. In addition, however, this type of tracing is complicated by the fact that the particular variable of interest may occur at several different points in the program; thus, complete internal tracing for a particular variable would involve a limited amount of backwards and forwards tracing at each point of occurrence of the variable in the program. It then appears that internal tracing is more complex than forwards tracing and may be more complex than backwards tracing, depending on the length of the path tracing for each occurrence and the number of separate occurrences examined.

An additional complication to data-flow tracing is seen when the overall structure of the program is taken into account. Larger programs almost always will be comprised of a number of separate and independent procedures, organized in various ways. These large programs will have a main program, consisting of some amount of internal code plus invocations of the other procedures -- "procedure calls"; each of the called procedures in turn may be likewise constructed. Information about the values of any number of data variables may be transferred from the calling procedure to the called procedure. Thus, data-flow paths will exist both *within* procedures and *between* them. When a data-flow path is completely internal to a particular procedure, all of the information is locally available in the code for the programmer to follow the path. However, when a path extends to other procedures, the code will -- at most --indicate the name of the called procedure and the name of the variable. To continue the tracing the programmer must search in the overall program for the code of the particular called procedure. Thus, tracing of data-paths involving called procedures introduces discontinuity into this process and is thereby presumed to increase its complexity. This type of tracing really involves tracing of both data-flow and control-flow, and it is considered separately following discussion of control-flow tracing below.

In summary, we have suggested three types of data-flow tracing within program procedures, with the hypothesized order of difficulty for programmers being: Forwards < Backwards < Internal. We also hypothesized that data-flow tracing *within* program procedures would be easier than tracing *between* procedures.

6.2.2.2 *Understanding control-flow* -- Three levels of control-flow may be identified: (1) transfer of control within the control structures of the programming language, (2) transfer of control between control structures (or to other types of statements) within the same procedure, and (3) transfer of control between different procedures (the special problems in tracing control flow for programs involving recursion or co-routines will not be discussed). Using PL/I as an example, the first type of control-flow, *intra-structure control-flow* occurs with the "DO", "IF-THEN-ELSE", and "BEGIN ... END" commands. Understanding of control-flow within such elemental control structures is attained by the learning of the programming language, and it offers no particular difficulties in understanding an unfamiliar program for modification purposes. However, when such structures are nested within each other, as in "IF X THEN IF Y THEN IF Z ...", it is useful for the programmer to have available a listing of the program which indexes the depth of nesting for any particular statement (these are noramlly supplied as a compiler option).

The second type, *inter-structure control-flow*, refers to transfer among code sections within a single procedure. The complexity of tracing between-structure control-flow depends to a major extent on the use in the program of "GO-TO" statements (either conditional or unconditional). When these statements are present, the normal linear flow from one command to the next adjacent one is interrupted; the programmer must then search the program for the line number or statement label indicated in the GO-TO to find the continuation of control-flow. In the present software development *zeitgeist*, however, so-called "GOTO-less" programming is advocated very strongly within a context of a disciplined technique called "structured programming". This technique calls for, among other things: (1) eliminating GOTOs and restricting the types of other control structures to three: IF-THEN-ELSE, BEGIN-END, and DO (including DO-WHILE and DO-UNTIL), and (2) requiring that the transfer of control between these structures be linear, with only one input and output path to each. This technique is now very widely adopted, and it appears likely that it will become the standard for almost all new software development efforts.

The implication, for control-flow tracing, of this trend towards structured programming is that *inter-structure* control-flow tracing for modification purposes will be greatly simplified in these new-style programs. Thus, with respect to tracing control-flow within a program procedure, the major difficulty expected is for *intra-structure* tracing within multiply-nested structures. In particular, nesting of IF-THEN-ELSE structures to a depth of several levels can lead to difficulty in determining the code-location to return to following completion of some internal path. However, even this difficulty may be removed, in view of the increasingly strong advocacy of restricting levels of nesting to two or three.

Thus, we anticipate that, unlike data-flow tracing, difficulties in tracing control-flow will be limited to the third type identified, *between-procedure control-flow tracing*. This type of tracing will become increasingly important due to a third aspect of the structured programming technique which calls for a "top-down" approach for software development. With this approach the programmer is enjoined to begin development of the program at a very high level, specifying a small number of sequential stages at this level, and then decomposing each stage into a sequence of more detailed stages, ultimately arriving at the final program details. The mechanism for accomplishing decomposition is to express each higher-level stage as a *call* to another, lower-level, procedure. Each of these in turn contains calls to procedures which refine the processing activity of the program. Typically, the ratio of procedure calls to the total code within any one procedure will decrease as the level of refinement increases, until ultimately the "lowest" called procedure will consist entirely of the basic types of command structures found within the programming language. For example, at the highest-level, a program PROG might consist of only three statements: "CALL INPUT, CALL PROCESSING, CALL OUTPUT." The INPUT procedure in turn might again consist of only

The 19 functional requirements were given to participants under various types of grouping relationships, but this variable had very little effect in either this or the next experiment. In

three statements, e.g., "CALL READ.IN.DATA, CALL CHECK.DATA.SYNTAX, CALL CREATE.INTERNAL.DATA", and so on.

In tracing the control-flow between procedures for such structured programs, three types of tracing may be identified, similar to the types for tracing data-flow. The first type, *lower-level control-flow tracing*, refers to the tracing of the paths from higher-level procedures to lower-level ones. Within any program procedure, only the next lower level is immediately apparent from the code of that procedure. To continue tracing it is necessary to find where in the overall program is listed the code for that lower-level called procedure. The ease of finding this code depends on the manner in which the code for the independent procedures is listed and identified. Nevertheless, lower-level tracing easily makes available *all* of the procedures called at the next lower level; continuation of tracing requires only selection of the desired called procedure and then again finding its code.

Higher-level control-flow tracing, is expected to be more difficult -- even more difficult than backwards data-flow tracing, to which it is similar. In this type of tracing the programmer does not have available an immediate context of occurrence of the procedure in some higher-level procedure; this lower-level procedure will be listed separately in the program, and the programmer must scan the other procedures for its mention. Since independent procedures are often written as separate files, *the programmer cannot easily use* computer editors to search a single file of all of the procedure listings for the desired procedure. Without any additional documentation, the programmer is best advised to use forward tracing, beginning with the highest-level procedure, and search the forward paths for the target, meanwhile keeping a record of these forward paths to construct the backward one.

The third type of between-procedure control-flow tracing, *procedure-occurrence tracing*, involves limited forwards (and backwards) tracing of all of the occurrences of a particular procedure. This can involve an exhaustive forwards tracing of all of the procedure calls in the main program and is therefore considered to be the most difficult.

In summary, for tracing control-flow, two types of within-procedure tracing and three types of between-procedure tracing have been identified. For tracing *within* a particular procedure, the relative difficulty of the two types depends on two factors: the level of nesting and the programming style. For non-nested programs written according to a top-down structured programming philosophy, neither type is expected to be at all difficult; for nested programs in this style, intra-structure tracing may be marginally more difficult than inter-structure tracing. For programs with GOTOs, inter-structure tracing could become mildly difficult (intra-structure tracing is not influenced by style factors). Tracing *between* procedures is expected to be relatively more difficult than tracing within them, with the three types ordered in increasing complexity as follows: lower-level, higher-level, and procedure-occurrence tracing.

6.2.2.3 *Tracing data-flow between procedures.* Two aspects of this type of data-flow tracing are discussed: (1) determining what variables are named as they are passed into or out of other procedures, and (2) determining whether the variables have been used or changed by the called procedure. These aspects are called "aliasing" and "use determination" and are treated separately.

Aliasing -- In the previous section an oversimplified illustration was given for hierarchical programs written according to a top-down "decomposition" technique. For example, one of the elements of the example's INPUT procedure was the procedure called CREATE.INTERNAL.DATA, suggesting the re-structuring of the input data into a form suitable for internal processing. In actual practice, the inputs and outputs of independent

5.2.4.4 *Software Design* Our last research project concerning design involved the actual design of a software system by experienced programmers. We asked eight programmers with

procedures cannot be merely suggested but must be explicitly specified. Using PL/I syntax, this specification might be given as follows:

(a). As called : CALL CREATE.INTERNAL.DATA (LIST1,LIST2);

(b). As defined : CREATE.INTERNAL.DATA: PROC (INDEX,NAME);

In (a) the procedure "CREATE.INTERNAL.DATA" is called from some other procedure, and two variables -- LIST1 and LIST2 -- are specified as part of that call. Given that use of "global" variables is avoided, these two variables will be defined and/or used somewhere in this calling procedure. Their specification in parentheses following the CALL indicates that the called procedure has been written to expect two variables to be given it. However, only the *structure* of these variables is fixed ahead of time, the *name* of the variable is not fixed -- it can be any named variable which has the correct defined structure. In (b) is shown the beginning of the code-definition of the called procedure, and the two variables it expects are here called *INDEX* and *NAME*. The programming language compilers are written to perform a one to one mapping of the variables specified by the calling procedure, as in (a), to those used in the code of the called procedure, as in (b), such that, for this example, the variable INDEX corresponds to LIST1, and the variable NAME corresponds to LIST2. Similarly, in another *procedure call*, the variables LIST3 and LIST4 may be mapped onto INDEX and NAME.

By the means described above it is possible to have a particular variable (and its associated data structure and values) identified by one name in one procedure but by also by other names in other procedures. With respect to any one procedure, the *other* names for a particular variable in other procedures are called "aliases", for example, the higher-level aliases of INDEX are LIST1 and LIST3. If the procedure in which INDEX is defined as an argument calls other procedures and passes INDEX to these, there could be lower-level aliases as well.

We believe that the tracing of data-flow between procedures, via the procedure call mechansims, is a necessary process for understanding programs and determining the best type and location of the necessary modifications. Even with the most advanced compilers, however, the alias information necessary for such tracing is not available in any convenient form. Extensive searching is required, which searching must include examination of the variable declarations as well as the code proper. We therefore believe this type of tracing to be the most complex of all the types previously mentioned.

Use Determination -- It is very frequently the case that variables passed to some called procedure are also returned by that procedure after it has completed its processing. From the point of view of program modification, we believe that it is very important for the programmer to determine *whether* (not necessarily *how*) the variables so passed and returned have (1) been used by the called procedure or, more importantly, (2) been modified by that procedure (are at least have had the opportunity to have been modified). Using the previous example, if LIST1 is passed into a called procedure as INDEX and subsequently returned, the first *usage-only* situation holds if INDEX occurs only as a control parameter in control structures (e.g., "DO X WHILE K LT INDEX ..."), as an index or pointer-variable for some other data structure (e.g., "MATRIX(J,INDEX) = TVAR"), or on the right-hand side of assignment statements (e.g., "TVAR = TVAR + INDEX"). The second situation, *modification-usage*, occurs when the variable appears on the *left-hand* side of assignment statements (e.g., "INDEX = TVAR + INDEX").

Determining whether variables are used or modified when passed is expected to be difficult to determine by code-inspection techniques. However, knowledge of whether passed variables are modified, vs. just used, could usefully assist the program-modifyer in embedding

differences appeared: PS used fewer adverbs, NS used a much wider variety of verb forms (e.g., present and past participles, more frequent use of modals like "should"), and NS used

the modifications in areas of code *in* which there is *little* modification of passed variables, thus avoiding undesirable side-effects. Thus, data-flow tracing between procedures -- checking for modification of passed and returned variables -- is expected not only to be very important for good program modification but also the most difficult of all types of tracing.

6.2.2.4 *Summary of postulated tracing processes* -- We list in the table below the various tracing processes discussed in the previous sections. The processes are given a relative index of expected difficulty, *D*, based on a 10-point scale, with 10 serving as the reference point of the most-difficult tracing process last discussed (see No. 5 in Table). The tracings are also given a ten-point index of expected importance for program modification activities, *I*, with a value of 1 indicating least, and 10 indicating most expected importance. These assignments are highly subjective, even though they were based on programmers' descriptions of their modification activities. The product of these two indices gives a reasonable weighted measure of expected importance and difficulty, *WT*.

NO.	TYPE OF TRACING PROCESS	PROGRAM TYPE	D	I	WT
	WITHIN-PROCEDURE DATA-FLOW				
1	Forwards input tracing	No Global Vars.	4	10	40
2.	Backwards output tracing	No Global Vars.	5	7	35
3.	Internal var'ble tracing	No Global Vars.	6	8	48
	BETWEEN-PROCEDURE DATA-FLOW				
4.	Alias Tracing	Top-Down Prgrms	9	9	81
5.	Use-Determination Tracing	Top-Down Prgrms	10	6	60
	WITHIN-PROCEDURE CONTROL-FLOW				
6.	Intra-structure tracing	Shallow Nesting	2	2	4
7.	Intra-structure tracing	Deep Nesting	4	4	16
8.	Inter-structure Tracing	Structured Prgms	1	7	7
	BETWEEN-PROCEDURE CONTROL-FLOW				
9.	Lower-level Tracing	Top-down Prgrms	5	9	45
10.	Higher-level Tracing	Top-down Prgrms	6	5	30
11.	Procedure-occurrence	Top-down Prgrms	7	7	49

These tracing processes, and their relative expected difficulties, provided the primary basis for the development of the modification tools, as discussed below.

6.2.3 *Considerations for tool development*

In the development of tools for subsequent testing, we took into consideration five factors: (1) the most heavily weighted (difficult and important) tracing processes (this was the factor most emphasized), (2) the choice of a specific programming language, (3) the type of

productivity or quality gains. Such conventions would include: (1) use of only imperative sentences (as opposed to declarative ones), (2) avoidance of embedded phrases or clauses, (3)

programs to be modified, (4) the information about programs which should be made available, and (5) the choice of representation-format for this information.

The *tracing processes* which we believed most needed supporting by modification tools are those in the table above having the highest weight. We noted that four of the five highest weighted processes involve tracing between procedures. We therefore determined that, whatever else might be decided, the modification tools should clearly show the data-flow and control-flow relations between procedures. Secondarily, we noted that two of the three within-procedure data-flow tracing processes were also highly weighted, and we decided they should also be portrayed in the tools.

Choice of a specific *programming language* was necessary since the specific details of the language would influence decisions concerning the remaining factors (we did not believe that program modification could be usefully studied using an artificial laboratory programming language). We chose PL/I as the target language because of : (1) its capability to comply with requirements of top-down structured programming philosophy (thus removing FORTRAN and APL from primary consideration), (2) its increasingly wide-spread usage for all types of applications (e.g., not just for business applications, for which COBOL and RPG are oriented), and (3) the sophisitication and power of its compilers.

We believed that the *programs* for developing and testing the tools had to have the following characteristics: (1) be large enough to benefit from the tools (e.g., at least 150 lines of code), (2) have several levels of depth of procedure calls (e.g., at least four levels of procedures calling lower procedures), (3) have a relatively large number of data variables (including input/output variables), differing widely in their structure, again to provide an opportunity for the tools to benefit (e.g., 10-20 variables within procedures, 10-20 variables passed between procedures, and data structures to include PL/I structures, arrays, vectors, and scalars), and (4) be of at least moderate complexity in terms of the processing algorithms within procedures (e.g., embedded control structures) as well as the transfer-of-control between procedures (e.g., calling the same procedures from different higher-level procedures). After checking several hundred PL/I programs, we decided our criteria would best be met by writing our own programs. We constructed four realistic programs which met all of the above criteria and, in addition, differed in the extent to which they conformed to the standards of top-down structured programming.

In considering what *information about programs* should be made available by the modification tools, we decided against supplying any information which was derivable only from an understanding of the documentation. Rather, we believed the information to be supplied should be restricted to that which was known about the program by the PL/I compilers. In this way we would be assured that this modification information could be automatically supplied, perhaps even on-line, without requiring advances in the state-of-the-art of compiler design.

After some examination of compiler characteristics, we chose the IBM 370 PL/I Checkout Compiler as the source for all modification information to be supplied. Although this compiler does not presently "tell" all it "knows" about a PL/I program, its internal processing provides sufficient bases for determining all of the characteristics of data-flow and control-flow. We chose to make available (in various formats, see below) the following kinds of information: (1) transfer-of-control relationships between procedures, including alias and use information, (2) the dimension and data-type of all variables, (3) the statement numbers, in the listing of the program, where each variable was defined as well as used, and (4) the "type" of use of each variable, in terms of nine classes (giving value, receiving value, control parameter in an IF statement, control parameter in a DO statement, an index of an array, a

purposes. Thus, for specifying a file-search process, the designer would ... with a description of the process sought -- e.g., a file-search --along with the particular

key in a PL/I structure, an argument or parameter in a procedure, an external input, and an external output).

At least equal in importance to any of the above factors is the choice of the *representation-format* for the modification information. The format should be easy to understand and use and should not obscure the relationships among the program elements; in particular, when there are "patterns" in the program code, these patterns should likewise be detectable via the modification information. Ideally, the format should be adjustable to the programmer's particular needs at the time, by means of suppressing irrelevant information; however, such a format would require on-line implementation of the modification tools. Since we did not feel sufficiently knowledgable about what kinds of information would, in fact, be useful, we restricted our format choices to a variety of off-line documents. These documents varied primarily in terms of the ways information was presented, and, secondarily, in the choice of information presented. Four basic types of formatting were used: (1) relatively unannotated directed graphs, particularly for showing data-flow relationships, (2) highly annotated flow-diagrams, particularly for showing transfer-of-control between procedures, (3) symmetrical arrays, with data variables as the row and column identifiers and with type-of-use information as the cell entry, and (4) relational tables, with the leftmost "key" column containing data variable names and the remaining columns containing various types of information.

6.2.4 *Modification tools tested*

We tried out a rather large number of different ways of choosing and representing information for modification purposes. Some of these were abandoned even before experimental testing. For example, we initially thought that a directed graph might be a useful way of displaying data-flow relationships. However, such a format is too cumbersome to display the necessary details of data-flow (or control-flow, for that matter), very rapidly becoming unreadable for programs of even quite modest size.

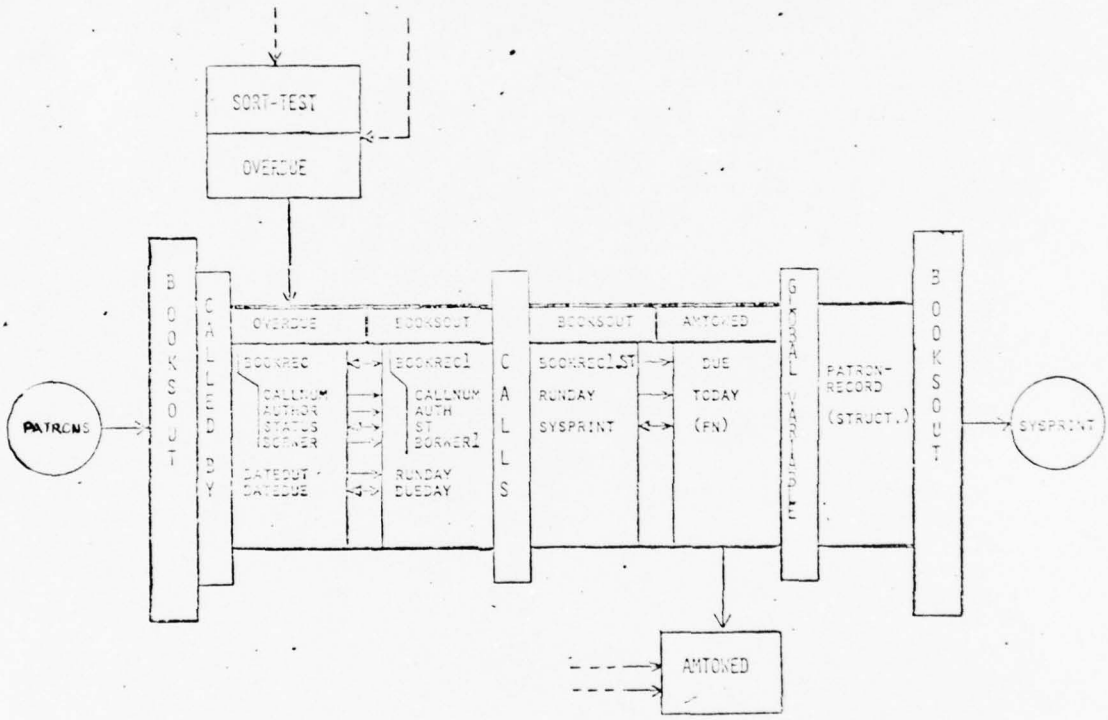
After several variations of the remaining three types of formats considered (annotated flow-diagrams, symmetrical arrays, and relational tables), we developed two types of information formats for representing the modification information which we used subsequently for experimental testing. The first used two types of representation, a flow-diagram for between-procedure and a symmetrical array for within-procedure information. In the second evaluation we used a single relational table for all information. These two methods of representing information are described in detail.

6.2.4.1 *First modification-information format* -- In this and the following section we will illustrate the formats using excerpts from the modification information actually tested. We do not present any of the program code, nor do we discuss the program itself in detail; such a presentation would require a very lengthy treatment and would contribute little to the present discussion of the tool formats. However, a brief overview of the program may be useful for reading the format information. The main program, called LIBRARY, maintains a record about books borrowed from a library. Aside from updating the files when books are borrowed or returned, the program provides a daily printout of overdue books (via the procedure BOOKSOUT) and the amount the borrower should be charged for these (via computation by the procedure AMTOWED). Other procedures print mailing notices to borrowers with overdue books, also listing the overdue fines (thus, AMTOWED is called from several procedures). The complete program has about 180 lines of code, 12 different procedures, and

```
1.1.2 DO UNTIL END OF FILE OR UNTIL "ERROR"
   1.1.2.1 GET NEXT WORD
```


a maximum calling-depth of 5 (i.e., at least one procedure -- AMTOWED -- is called by another procedure, which itself is called in turn, this continuing up to a total sequence of four higher-order calls). This first type of representation involved two different types of formatting of information, one for between-procedure relationships, and one for within-procedure relationships. These are discussed separately below.

6.2.4.1.1 *Between-procedure information.* The representation of between-procedure information in this first type of format involved a type of annotated flow-diagram, as shown below.



There are four major entities to the diagram, as indicated by the different figure shapes. The largest figure contains detailed control-flow and data-flow information about a particular procedure and its relation to other procedures; it is identifiable by the long and narrow vertical rectangles on each end, containing the name of the procedure (BOOKSOUT, in this case). The rectangles above and below indicate the procedures which call BOOKSOUT and which are called by BOOKSOUT -- OVERDUE and AMTOWED, respectively. The occurrence (and location) of dotted lines into these rectangles indicates that these procedures are in turned called by (or call) other procedures. Further, a horizontal line within a rectangle indicates that a so-called "internal" procedure is contained within the main procedure. Thus, OVERDUE is seen to be an internally-defined procedure of the external procedure SORT-TEST (the highest name within a rectangle is always the name of the external procedure; subsequent names are internal procedures). The circles indicate external input (PATRONS) and output (SYSPRINT) from the highlighted procedure BOOKSOUT. A similar diagram is available for each of the other procedures, always showing the calling structure by rectangles and arrows above and below the main figure, and showing the external inputs and outputs (if any) on the left and right, respectively. In addition to these diagrams a broader but

less detailed view of the overall calling (and input-output) relationships is shown by a separate diagram consisting only of rectangular figures and their arrow flow-indicators.

The internal structure of the main detailed figure has three components. These components are separated by narrow vertical rectangles labeled, from left to right: CALLED BY, CALLS, AND GLOBAL VARIABLES. In the CALLED BY component there may be one or more two-part segments whose left-hand portion provides information about the procedure which *calls* the detailed procedure, and whose right-hand portion provides information about the detailed procedure itself. Thus, the headings in the CALLED BY segment of the BOOKSOUT diagram are OVERDUE and BOOKSOUT, respectively. Although BOOKSOUT is called only by the OVERDUE procedure, had there been more, there would have been a separate two-part segment for each.

The information contained within the two-part segments specifies the names of the *variables passed into the procedure, as they are called in the higher-level procedure,* together with their names as assigned in the called procedure. Thus, the variable BOOKREC occurs in the procedure OVERDUE and is passed to BOOKSOUT, in which it is called BOOKREC1; similarly, CALLNUM has only one name in both procedures, while DATEOUT becomes RUNDAY. The bent vertical line on the left of BOOKREC down to BORWER (for "borrower") indicates that all five variables so enclosed are related and constitute what in PL/I is called a *structure* BOOKREC is the name of this structure, and it has four components which may (and do) have different data-structures (as CALLNUM is an alphanumeric character vector, and STATUS is an integer scalar). The final feature of the CALLED BY component concerns the arrows *between the two parts of the segment, having either a small* arrow on the right or both a small right-end arrow and a large left-end arrow. The single-headed arrow indicates that the variable name on the left is used in the called procedure by the name shown on the right. The double-headed arrow indicates not only the above name-mapping but also the fact that the called procedure passes back to the calling procedure a *new* value to be reassigned to the variable name on the left. Thus, DATEOUT is assigned to RUNDAY, but DATEOUT does not receive a new value. However, DATEDUE is passed to the BOOKSOUT procedure (becoming DUEDAY) which subsequently returns a new value for DATEDUE.

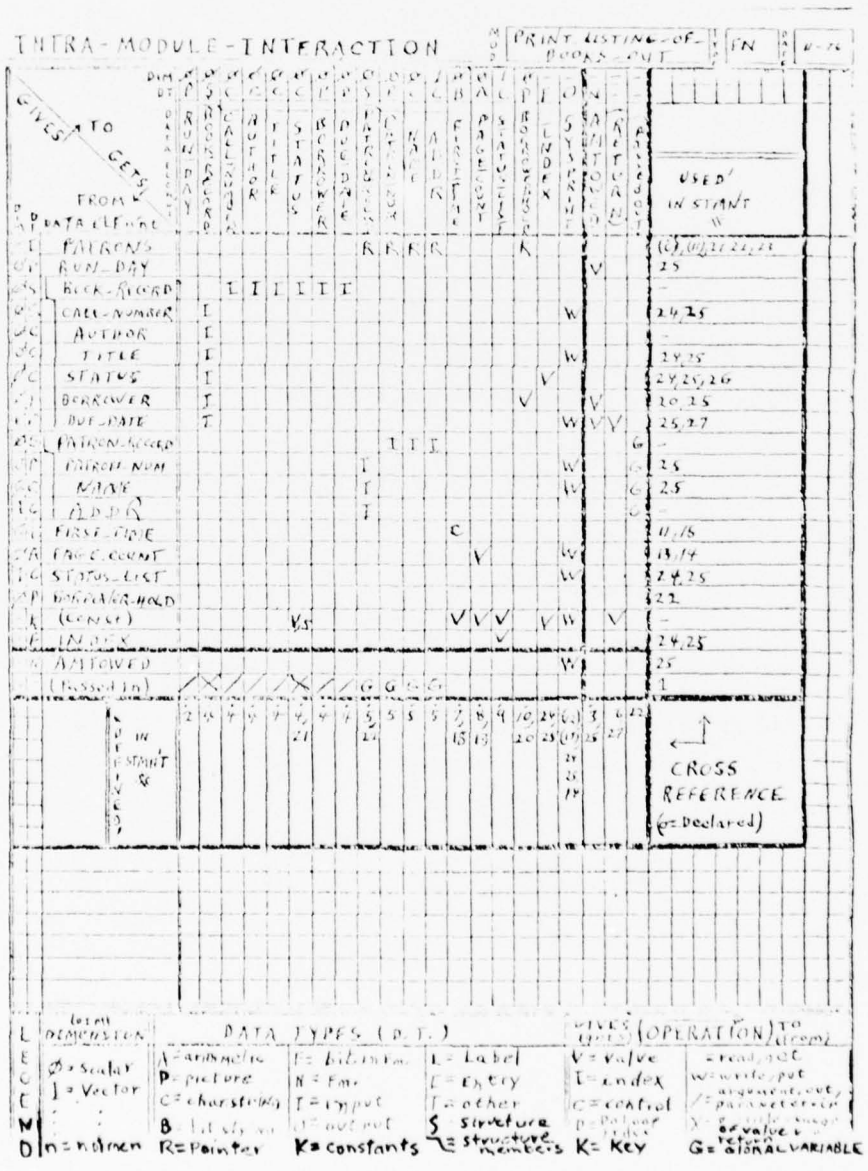
The second component, the CALLS component, gives information about procedures which are called by the procedure being detailed (in this case, AMTOWED). The information within these segments is constructed and read exactly as for the first component (e.g., the ST part of the BOOKREC1 structure, BOOKREC1.ST, is passed to AMTOWED when called by BOOKSOUT, but becomes the variable DUE in AMTOWED; this value is not modified and re-assigned to BOOKREC1.ST upon completion, however).

The third component, GLOBAL VARIABLES, contains the names (and data-type) of all variables which can be used by the procedures in the figure without being specifically passed into or out of the procedures; thus, PATRON-RECORD is a PL/I structure and can be used by all four procedures in the diagram.

It should be noted that the use of global variables, as well as entry points for internal procedures (e.g., as indicated by the arrow pointing into the OVERDUE procedure), as well as internal procedures themselves, are all nowadays considered not to conform to the best programming standards. Nevertheless, we designed our modification-information formats so that they could accommodate *all* of the characteristics of existing PL/I programs, not just ones written according to the best style criteria; for these latter, the GLOBAL VARIABLES component and the internal segmentation of rectangles (to indicate internal procedures) would simply be omitted.

This type of flow-diagram described above thus represents all of the between-procedure information specified earlier as being important for assisting the program-modifier to come to an adequate understanding of this aspect of the to-be-modified program, via control-flow and data-flow tracing between procedures.

6.2.4.1.2 *Within-procedure information.* The information about data (and control) relationships within procedures was portrayed using a symmetrical array format in which all of the data variables within the procedure were listed as column and row headers; the "cells", formed by the interaction of rows and columns contained characters which indicated the nature of one variable's relation to another. An example of this type of format is shown below.



Forward data-flow tracing of a variable is accomplished as follows: locate the desired variable as the identifier at the far left of one of the rows; then read across the row and find the non-empty cells containing some letter; read up to find the column identifier at the top of those columns; these are variables that receive value from the row variable, in the manner indicated by the letter in the cell; the nature of the value is indicated by the legend in the lower right-hand corner of the diagram. Backward data-flow tracing is accomplished by reversing the process, beginning with a *column* variable, finding the non-empty cells, and reading left to determine the variable giving value. This manner of reading the table is cued by the information given in the upper right-hand corner of the figure. Thus, for forward data-flow tracing, the figure indicates that the variable PAGE__COUNT gives value to itself (e.g., "PAGE__COUNT = PAGE__COUNT + 1") and also is written as the output variable SYSPRINT. The variables listed in the rows and columns are further partitioned by heavy lines to separate variables which are passed into or out of the procedures from those which are strictly internal. For example, the row variable RUN__DAY can be seen to be passed out as a value of a parameter in the function AMTOWED. (Note that the concept of a "variable" is stretched to include both functions -- N --and built-in functions -- F).

The topmost line of the figure shows three fields of identifying information: (1) the name of the module (e.g., "PRINT LISTING OF BOOKS OUT"), (2) its type -- external or internal procedure, or function (e.g., FN), and (3) the date of generation of this figure (e.g., 11-76). Three types of information are also supplied for each variable in the figure: (1) the dimension (DIM) and data type (DT) of each variable is given at the left (top) of each row (column) variable (the entries are defined on the left side of the legend, at the bottom); (2) all of the places in the program where the variables are *used* are indicated by the statement numbers on the right of the figure; and (3) all of the places in the program where the variables are *declared* (i.e., defined) are indicated by the statment numbers at the bottom of the figure (above the legend).

This figure thus contains all of the information typically available from PL/I compilers in the form of cross-reference lists, plus, of course, much more information about the interrelationships among the data variables. The formatting of the figure as a symmetrical array, with the same variables as both row and column headers, provides for the detection of familiar "signatures" of program code. Thus, the pattern of a column and a row of "I"s as cell entries, joining in the upper left-hand corner indicates a PL/I structure (e.g., BOOK__RECORD). similarly, a string of "R"s in a row indicates a read-in assignment of values to an internal data structure (e.g., PATRONS). Similarly, other patterns can be recognized as indicated certain types of code, as several "D"s or "C"s will indicate iterative and condition-testing procedures, respectively.

6.2.4.2 *Second modification-information format* --While the first format can be used quite effectively for forwards and backwards data-flow tracing within procedures, it does not directly provide for easy internal variable tracing within procedures. Also, that format involved two different representations, one for between-and one for within-procedures. For these reasons, we developed an alternative format which combined all of the information into one figure. An example of this type of format (simplified and somewhat contrived) is shown following:

PROCED: MLIST

Variable Name	DECLD: TYPE	DECLD: DECLD	DEFINED: OPERN	DEFINED: GETS VALUE FROM	DEFINED: STMT#	DEFINED: PASSED FROM: MOD	DEFINED: PASSED FROM: ALIAS	USED: PASSED TO: MOD	USED: PASSED TO: ALIAS	USED: PASSED TO: STMT#	USED: INDEX/KEY/QUAL: OPERN	USED: INDEX/KEY/QUAL: VAR	USED: INDEX/KEY/QUAL: STMT#	USED: CONTRL	USED: STMT#	USED: OPERN	USED: GIVES VALUE TO	USED: STMT#
OPER	C	2	R	SYSIN	5			MSG	NB	9								
MATRX	A	2				LIB1	MAT				I←	X	7			V→	INV	7
											I←	Y	7					
INDEX	A	3												D	6			
X	A	3									I→	MATRX	7					
Y	A	3									I→	MATRX	7					
INV	A	3	V←	MATRX	7			LIB1	KEY	10								

This format, corresponding to a relational table, list all the variables within a module in the leftmost column. The two major sections of the table indicate how the variable is *defined* and *used*. The three ways in which a variable recieves value are: (1) in a declaration (here, only the data-type and dimension are given), (2) in an assignment statement or external input (the "GETS VALUE FROM" segment), and (3) as passed from a higher-level procedure (the "PASSED FROM" segment). The four ways in which a variable is used are: (1) as passed to a lower-level procedure (the "PASSED TO" segment), (2) as an index, key, or other type of qualifier of some data structure (the "INDEX/KEY/QUALIFIER" segment), (3) as a control variable in a DO or IF control structure (the "CONTRL" segment), and (4) in an assignment statement (the "GIVES VALUE TO" segment).

For example, the table indicates that the two-dimensional array MATRX is passed into the procedure MLIST from the higher-level procedure LIB1 (known there as MAT). MATRX is indexed by two variables, X and Y (in statement 7), and it gives value to the arithmetic scalar variable INV. This variable INV in turn is passed back to the calling procedure LIB1 where it is to be called KEY.

6.2.5 *Behavioral Findings*

Our evaluations of the usefulness of the program modification tools described above were conducted under the following conditions:

1. Experienced PL/I application programmers served as volunteer participants in the experiments..

2. Moderately complex PL/I programs were used.

3. Participants worked at IBM 3277 display terminals connected to an IBM 370/168 VM interactive system.

4. In the major control condition, the participants solved program modification problems using information facilities from existing PL/I compilers; this output included the source listing, attribute table, and cross-reference list.

5. In the major experimental condition, the participants solved program modification problems using the program modification tools described above; in some cases participants were also supplied with the above compiler information about programs.

6. Participants given the modification tools were trained in their use by means of a package of training materials given to them the night before the experiment, or at least 2 hours prior.

We used two tasks for evaluating the effectiveness of the modification tools, as described separately below.

6.2.5.1 *First Testing Task* -- In our initial task for evaluating the modification tools, we gave 14 programmers (7 trained with the tools and 7 given the compiler information) a modification to be made to the program and asked them to actually insert the necessary code and debug their modified program. For example, in terms of the library-program example (see Section 6.2.4.1), we asked that the program be modified such that overdue book fines doubled for every extra week the book was not returned, and warning letters were to be sent to the borrower after the second week; participants were to actually modify the program code to accomplish these new performance aspects. The performance measures chosen to index the effectiveness of the tools in this task were: overall correctness (does the modification accomplish the specified requirements), overall time to complete the modifications, and number of separate modification activities required to accomplish the modification -- e.g., the number of editing or other commands issued to the computer system to enter, check, and debug the modifications.

The problem with this task was that the details of using the computer editors obscured all other aspects; participants spent sizeable portion of time manipulating the program in the editor in ways that were not necessarily related to actually making the modifications -- e.g., reviewing the program on-line instead of as requested from the printed listing, using the editors' search facilities to do data-flow tracing, making modifications to other aspects of the program not necesarily involved in the task (but which aspect did not conform to the participants' sense of proper programming). Even when participants carefully planned the modification to be made off-line and used the editor only to make the changes, still there was considerable editing "noise" which made any measure of overall time or number of editing steps not good indices of the modification task -- and the effectiveness of the tools. For example, participants would make an error in where they modified or inserted new code, and many subsequent editing steps would then be needed to correct this error (particularly, when it was discovered some time later after entry); similarly, simple typing errors accounted for many steps and minutes of correction time. In order to insure some level of quality of the programs, *after* modifications were made, we asked that participants submit their programs to the compiler and perform any necessary debugging to eliminate any instances of a class of what are called "serious errors" detected by the compiler. Quite frequently participants did, in fact, have these serious errors, and subsequent debugging often required more time than the initial planning and insertion of the changes; thus, this phase introduced additional noise into the overall measures of time or actions to complete the modification.

This method of testing the tools can therefore be seen to be undesirable, at least for the initial exploratory evaluations; not only was an excessive amount of participant time required (an average of 3.5 hours) but the data is confounded by irrelevant editing activities, and the

level of correctness (about 50 percent whether tools were used or not) was too low, indicating that the task was too difficult. No significant performance differences between the tool and no-tool conditionswere found for any of the performance measures.

6.2.5.2 *Second Testing Task* -- After the above experience we decided to change the task and eliminate the contribution of computer editing to the measures of modification performance. In the second task, we asked participants to answer in detail a set of 15 questions regarding the programs and the proposed modifications. Each of the questions were intended to address some component aspect of the overall modification task -- e.g., understanding the overall characteristics of the program, locating the code containing variables mentioned in the modification description, determining the calling and alias relationships for procedures to be involved in the modification, etc.

Experimental Design Each subject in the questionnaire experiments answered two sets of questions, one with and one without the program modification tools; each set of questions involved a different PL/I program, matched for relevant complexity features (such as number of modules, calling structure, number of variables, etc.), but differing, however, in the nature of the operations performed and the data structures used (the so-called "library" program involved considerable operations on character-strings -- matches, searches, etc. -- and used PL/I character data-structures, as well as PL/I "structures"; the second program, providing a set of mathematical calculating procedures, employed arithmetic operations and array data-structures). A counter-balanced design was used to insure that there were equal occurrences of, e.g., the different programs being seen first or second, the association of modification tools with specific programs, the occurrence first or second of use of the tools, etc.

There were three "mini" experiments which differed in terms of the type of information given along with the modification tools -- the so-called "tool-condition". In the first experiment, participants in the tool-condition received the modification information as represented in the *first* format (see Section 6.2.4.1); in addition, they also received all of the compiler information given in the "no-tool" condition. In the second and third mini experiments participants in the tool-condition received the modification information according to the *second* format (see Section 6.2.4.2); in the second experiment they also had the compiler information available, as in the first; but, in the third experiment, only the modification information was available -- not even listings of the programs were seen. From four to eight participants were used in each experiment.

These experiments have too few participants to permit any strong conclusions; however, we believe that this pilot work involves *enough* participants for detection of promising directions for larger-scale experiments.

Results -- A "correctness" score of from 1 to 10 was assigned to the answer given each question; in addition, the total time to complete each question was measured and recorded. In terms of these measures, the findings from the three experiments were quite consistent: performance was never *worse* in the conditions where the modification information was given -- even when that was the only information. In experiment 2, using the second information format with compiler information, the performance was marginally superior with the modification tools than without ($p<.07$). There were no significant differences between performance with thefirst modification-information format vs. the second type of format.

These findings provide rather strong support for the effectiveness of the modification tools. The reasonable expectation for these experiments is that performance with the tools should be worse than that with the compiler information, for several reasons: (1) participants had years of experience in using the compiler information; (2) they had at most a few hours

experience in using the tools; and (3) the tools were complex and not easy to learn. That performance was never worse with the tools strongly suggests that the tools were in fact highly appropriate for supporting modification activities (at least those tested by the questionaire).

Participants provided a number of recommendations concerning the tools. The most common suggestion was to make the tools -- whether in the first or second format -- available on-line on a selection basis (i.e., let the programmer specify the variables of interest, suppressing the remaining information). In addition, however, they recommended that the modification-information be supplied with all programs as part of the documentation.

6.3 *RECOMMENDATIONS:*

The excess costs of program maintenance justify immediate action towards providing tools to facilitate the modification process. Our first and strongest recommendation is that the off-line tools developed in our pilot research work be evaluated in a full-scale experiment, with programs of even greater size than we used. In particular, we believe that the second information format used may be the most effective, even if used only for *within-procedure* information.

Secondly, we recommend that the utility of providing such tools on-line, for interactive use, be evaluated. These tools should, at *minimum*, provide for the step by step display of forwards or backwards data-flow tracing. They should also support general questions about data and control flow (not in natural language, of course, but in some appropriately constrained syntax). Example of such questions are: "Could the variable VARS ever be returned in a modified form by some procedure to which it was passed?" (example of an answer: "Yes, VARS is passed to procedure PCNT in which it can be modified and returned; do you want to see the details of this?"); or "What procedures or functions are the most frequently called by other procedures?"; or "What procedures have no calls to other procedures?".

7. GENERAL BEHAVIORAL ISSUES WITH INTERACTIVE SYSTEMS

7.1 *MOTIVATION:*

Programming performance is affected not only by the conceptual nature of the task and the formal aspects of the programming language, but also by the complex computing environment in which the programming is embedded. These factors include such items as the mode of programmer-computer interface (batch vs. interactive), the availability of program debugging and testing tools, computer system performance (e.g., response time, data-access times), characteristics of editors for creating and modifying programs, etc.; all of these can greatly influence both the quality of programs and programmer productivity.

Accordingly, we have continuously monitored and kept ourselves informed of studies in the human-computer literature which relate to behavioral issues in programming. As a conclusion to our other, more experimental, contract work we believed it to be important to communicate this broader range of information in the form of a review (see section 7.2.4).

We also believed that it would be useful to communicate the broader conceptual views that we have developed during the contract period concerning the *general* behavioral issues in programming, program development, and ultimate program usability.

7.2 *SUMMARY OF WORK:*

7.2.1 *Overview:*

Our contractual work, and our review of the relevant literature under the contract, has led us to the belief that the behavioral issues in program-development and user interfaces are best organized from three perspectives, all relating to what is known about the application to be programmed and used.

There are a number of general behavioral issues which cut across all types of applications and relate to the broad characteristics of the computing environment, such as the performance, facilities, and user-interfaces of computer systems . This is one perspective (12). Two other perspectives derive from a two-part categorization of programming applications into those which support *routine tasks* as contrasted with those which support *problem-solving* tasks.

Our review was limited to that of the first-mentioned perspective, which work is summarized in 7.2.4.

7.2.2 *Routine-task applications:*

We present here our conceptualization of "routine" tasks to better define both the potential and the method for developing programming applications for these tasks. This view stresses the behavioral issues involved in the implementation of computer programs for these tasks.

7.2.2.1 *Characteristics of routine tasks* -- We define tasks as being "routine" when they possess all of the following four characteristics. Routine tasks --

1. are triggered by a small number of clearly-defined external event patterns

2. primarily involve the processing of external data, which processing involves highly observable, structured, and predictable intermediate and final output.

3. are governed by explicit, highly available, highly structured, primarily linear, procedures, involving only low-level decision-making and little personal exception-handling.

4. involve a short trigger-to-completion cycle (in terms of either duration or discrete steps)

Of this set of human activities so-defined above, we exclude from consideration those tasks involving the human as a controller in closed, short [illegible], feedback loops such as in tracking, precise machine operation, etc. What remains, however, appears to be by far the most common type of activity for the largest number of people (at least in industrialized societies). Included are almost all tasks involving *prepared forms* (e.g., questionnaires, applications, reservations, orders) as well as the majority of other business and governmental bureaucratic activities.

7.2.2.2 *Stage model of Routine Task Activities* -- We now present a generic model of the sequential activities involved in (non-control) routine tasks to provide the basis for a set of recommendations intended to improve usability as well as more effective computer-implementation of these applications. The routine task model involves five steps, with the possibility of recursive invocation of the model in steps 3 and 4.

1. *Pre-screening For Potential Trigger Event* -- This step involves ascertaining whether either (1) the data presented or (2) the queried goals of the agent involve features of the acceptable trigger-patterns. If not, the data or agent are referred elsewhere.

2. *Recognition of Trigger Event* -- Here the situation is examined carefully to identify an appropriate event-trigger, these being a small number of classes of situations for which the task in question has procedures to govern further processing.

3. *Procedure-Selection and Exception-Detection* -- Normally in these tasks it is a simple matter to identify the procedure appropriate to the trigger event. However, difficulties may arise whenever (1) the pattern is appropriately matched to more than one (possibly competing) procedure, or (2) the pattern, upon examination, is deficient or anomalous in some requisite characteristic. Such difficulties constitute *exception conditions* and require a decision before proceeding. Such decisions usually involve the execution of an information-gathering procedure either directly or indirectly by reference to a higher authority. The output of the decision is either termination of the procedure or identification of an appropriate pattern-procedure match. To the extent that the decision/information-gathering procedures invoked are themselves routine tasks,

exception-handling involves a recursive invocation of the model at this point. (If these decision-making or information-gathering tasks are *not* routine tasks, then this model is considered to be suspended and a *problem-solving* mode invoked; see Section 7.2.3).

4. *Initiation of Procedure* -- The procedure appropriate for the situation is now identified and begun. The procedure will typically involve a number of intermediate steps, primarily of data-input and data-combination forms. With a single exception each such step may be conceptualized as a recursive invocation of the model, with the data being pre-screened and recognized as in steps 1 and 2, mapped to the correct procedure and examined for exception conditions as in step 3, and the appropriate procedure begun in this present step 4. The exception to this recursion cycle is when the procedure identified involves only a transfer of data, intact and without modification, from one location to another (e.g., writing down a previously-checked orally-given piece of information, copying some information from one part of a form to another). In such cases the procedural steps are executed as given without recursion.

5. *Output* -- All data required by the procedure to be output are appropriately formatted, stored, and transmitted to the appropriate recipients.

7.2.2.3 *Behavioral issues in Routine Tasks* -- To the extent that the above is a useful characterization of actual routine task execution, a number of behavioral issues can be identified relating both to the development of computer applications to support such tasks as well as to the usability of those applications. These issues include:

1. *Event-trigger pre-screening and recognition* -- Task facilitation could be achieved by making available for reference those features of the acceptable triggering events which, when present in the situaton, assure that the particular routine task should be executed, or, when absent, indicate that the task should *not* be executed. The identification of the appropriate sub-procedure could also be similarly facilitated by providing a means for entering the feature-values of the trigger event and obtaining the appropriate sub-procedure to be executed (e.g., the identification of the appropriate form to fill out).

2. *Procedure availability* -- In situations where the task involves many different sub-procedures, time and errors in task execution could be reduced by making available to the person performing the task the step-by-step details of the procedure, particularly for steps in which errors or forgetfulness could be expected.

3. *Exception-handling* -- Most exception conditions involve the value of some particular datum not being included in a set (or range) of allowable values. These value-restrictions may be multiple and complex, and they may easily be forgotten or else remembered in error. Automatic checking of data-values for exception conditions could speed task execution as well as reduce user errors.

4. *Place-keeping* -- For tasks involving the possibility of recursively invoking subprocedures assistance could be provided by automatically keeping track of the hierarchical status of suspended, present, and remaining tasks, and making this record

available for reference. Further, provisions could be made for automatic transfer of relevant data up or down the hierarchical procedure-call structure to minimize redundant manual data-input (and thus minimize errors).

5. *Monitoring for error-control and automation-potential* -- Logging of time-spent and errors made during execution of the steps of a task procedure, over an extended period, (both by automatic and manual means) could provide a basis for determining error-prone and time-consuming aspects of the task procedure. Many such aspects necessitate human participation in the task (e.g., as a speech-understanding device, data-transfer mechanism, etc.), and specialized facilities could be provided to improve performance. However, for those problem aspects which do not require human involvement, prime opportunities for automating task components can thereby be identified. In particular, situations involving complicated computations or well-defined but multi-component exception-judgments are ideal candidates for automation.

In summary, we have provided in this section (1) a basis for identifying activities and programming applications which are *routine* in nature, (2) an outline of a plausible model of human behavior in such tasks, and (3) examples of behavioral issues involved in routine-task execution which can improve the effectiveness of program development as well as the usability of computer-supported routine-tasks.

7.2.3 *Problem-solving applications:*

7.2.3.1 *Characteristics of Problem-Solving Tasks* -- This is our second perspective, and these tasks differ from routine tasks in *each* of the four types of criteria used for defining routine tasks. In contrast to this former class of tasks, which required all four criteria to be present, the presence of *any* of the following is considered sufficient to define a problem-solving situation. Problem-solving tasks --

1. can be triggered by a large number of poorly-defined event patterns, both external and mental.

2. may include the processing of both external and mental events with little, or ill-defined, intermediate output, and with possibly unpredictable final output.

3. are governed by typically idiosyncratic procedures, which procedures, typically: are not explicit, are neither highly available nor structured, involve extensive use of context, and require a high degree of complex decision-making and personal attention to exception conditions.

4. indeterminately long trigger-to-completion cycles.

In contrast to routine tasks, which required all four criteria to be present, the presence of *any* of the above is considered sufficient to define a problem-solving situation.

7.2.3.2 *Taxonomy of Problem Types* -- As a result of our larger interests in complex cognitive behavior, which includes our contractual work, we have been developing a taxonomy of *types* of problem-solving. Our goal is to define a classification structure which: (1) would permit recognition of any particular situation as being an instance of one primary problem type, and (2), once so identified, the problem type would imply a set of characteristics unique to that type (e.g., useful solution algorithms, optimal information-display characteristics, data-humanagement facilities, processing aids, etc.). For example, for *Diagnosis* problems, one useful fault-location algorithm is to conduct tests which result in the elimination of roughly half of the remaining trouble possibilities after each test. Since we consider that many different problem types may be invoked as subproblems within each type, we also hope to specify the set -- and probable sequence -- of problem types likely to be invoked under each primary type. Again using the example of Diagnosis, a primary information-display requirement is to provide information on how the entity being diagnosed *should* appear, via, e.g., schematics, design specifications, etc. We further intend the classification system to apply equally well over variations in the nature of (1) the agent doing the problem-solving, and (2) the problem environment. Thus, medical diagnosis of animal disease, computer diagnosis of syntax errors in programs, and clinical self-analysis all should be characterized as Diagnosis problems without classification difficulty.

At this point we have not yet met all of these requirements for the taxonomy. However, we do have an interum set of 12 types of problems which we believe to be a useful basis for discussion. As all of these problem types are encountered in programming and program development, they can provide the basis for developing human-computer environments tailored to support specific programming task activities. We offer the results of this continuing work here to illustrate the kinds of problem-solving activity that could be supported in the optimized scenario we present at the end of this section.

Any one of six entities may be the focus for the problem-solving activity, based on the following orthogonal distinctions: single element vs. a collection of elements composing the entity, organized vs. unorganized elements -- holding only for collection --and active vs. inactive elements. (In these terms, a "system" refers to a collection of organized *active* elements, whereas a "structure" refers to a collection of organized *inactive* elements). Unless otherwise stated, the problem-type applies to all six types of entities.

The problem types are:

1. *Design* -- creating a representation of some entity which is ultimately to be built, which representation satisfies a set of functional requirements as well as restrictions concerning development time, allowable resources, unacceptable intermediate steps or final characteristics, etc.; however, many factors are ill-defined, such as the form of the final product, the development techniques, and the working principle.

2. *Cross-domain equivalency transformation* -- transforming the representation of an entity from one domain into another, which domains may have different properties, constraints, and environmental characteristics, but the new representation fulfills specified invariant and isomorphic relations of structure and performance with respect to the original; in particular the *organizational* and *interactive* relations tend to be preserved from one domain to the other. The transformation operations are not necessarily specified nor bounded; e.g., re-writing a program from one language into another.

3. *Mapping* -- specifies one of three simple assignment relations between two different sets of elements: one-to-one, one-to-many, or many-to-one. This is the degenerate form of cross-domain equivalency transformation, used primarily for unorganized or

non-interactive collections of elements; e.g., assignment of social security numbers to people.

4. *Assembly* -- the application of sequencing or binding operations to the parts of an entity as guided by some governing procedure so as to achieve unification of the entity in accordance with its prior design (applies to entities which are, or are to become, organized collections of elements).

5. *Operation/Manipulation* -- achieving control over the entity's movement in accordance with (1) the purpose for which it was designed, or (2) the instructions of any procedure.

6. *Understanding* -- determining (1) the design purposes fulfilled by the entity (if any), (2) the nature of the input events to which the entity is responsive -- as well as indifferent --(3) the nature of the events which the entity can produce as output, (4) the method of construction, (5) the general principles of operation, and (6) the specific means by which input and other events are combined or transformed to achieve the output. Understanding may be *partial* in the sense that not all of the above are determined.

7. *Test/Evaluation* -- these operations are performed by comparing an entity's performance, or structural or formal characteristics, either to some set of derived expectations (e.g., standards) or else to some other entity to determine the nature and degree of correspondence on some similarity metric (an important example being that of formal equivalence).

8. *Formal symbol-manipulation* -- replacing, modifying, or adding to the elements of a collection of well-formed logical propositions (e.g., formal grammar rules, logical calculus expressions) by means of the application of a pre-defined and closed set of allowable transformations; e.g., proving theorems (applies to collections of inactive elements).

9. *Rule induction* -- derivation of a generalization concerning classes of entities which states, with an arbitrary precision level, the collection of characteristics that can be expected to be present (or logically true) given another set of characteristics concerning the same entity classes; *structural* inductive rules specify only the features of the entities which must be present; *process* inductive rules specify the process steps that must be applied to the entities to obtain the predicted characteristics.

10. *Diagnosis* -- begins with the detection, via test/evaluation operations, of important discrepant characteristics, usually discrepancies between an entity's present state and the state it should have been in according to its design or prior known state; identification is made, to a variable degree of resolution, of the elements or the element-interactions, which produce the discrepant symptoms; e.g., program debugging. (The action of *repair* following successful diagnosis is considered to be a variant of an Assembly task).

11. *Information-seeking* -- the search for particular characteristic(s) of an entity or class of entities which fulfill a stated relationship to each other or to some other reference set.

12. *Choice Decision-making* -- selection of one alternative from a set of possible ones on the basis of a rule (decision function) which assigns values (relational or absolute) to each possibility and determines the maximum- or minimum-valued alternative as preferred.

7.2.3.3 *Optimized computer support of Problem-Solving* -- In a truly optimized human-computer problem-solving system we would envision the following kind of scenario: (1) The user and system conduct a dialogue to determine which of the above types of problems the user needs to solve; (2) a second dialogue ensues to develop a precise formulation of the problem, especially the functional requirements; (3) the system then establishes the optimal support environment for the problem (e.g., selection of the functions, data structures, and display parameters optimal for the particular problem-type; (4) via user-request, or system-recognition of the need, a sub-problem is invoked, involving a recursive execution of this scenario, and the system performs the necessary recording of all pertinent information at this level and then begins from step (1); (5) the system keeps track of all sub-problems, permits easy transfer of information from one sub-problem to another, and permits the user to re-invoke any prior environment as well as a new one.

7.2.4 *All applications:*

We have considered behavioral issues in programming from two perspectives, based on the type of task -- routine tasks and problem-solving tasks. Our third perspective concerns behavioral issues in programming and program development which cut across *all* applications, whether routine or problem-solving. Our substantive contribution here is the topical organization of behavioral issues as they relate to aspects of computer systems and a review of the work providing data or theory on identification and resolution of behavioral problems (12).

7.2.4.1 *Content of the Review* -- Published research was reviewed according to the following organization of topics (12):

1. Overview of behavioral-issue perspectives
2. System Characteristics
    2.1 Performance (e.g., response time, availability)
    2.2 Facilities (e.g., command languages, editors, file manipulation, data manipulation, inter-user communication, recovery philosophy)
3. Interface Characteristics
    3.1 Dialogue style
    3.2 Keyboards
    3.3 Alpha-numeric displays
    3.4 Speech input/output
    3.5 Graphics

7.2.4.2 *Key Behavioral Issues* -- Out of all the issues addressed in the review, we identify three which seem to us now as having the greatest potential for facilitating general behavioral usage of computers: Editors, File Manipulation, and Information-Partitioned Displays. We identify below the critical behavioral deficiencies in each area, augmenting the discussion provided in the review.

7.2.4.2.1 *Editing:* The nature and availability of editing facilities is by far the most important area. Not only is editing the most dominant activity in computer usage -- including program development -- but it also provides the greatest opportunity for improvements which can greatly facilitate work productivity and quality. Effecting these improvements requires only further behavioral analyses and not any technological advances; that is, the improvements can be implemented by providing enhanced editing-function software, even by extension of present

editors with edit macros written in the language of the computer system (e.g., IBM's CMS EXEC language).

There are two problems which underlie what we see as the key editing difficulties: (1) what the user wishes to do maps only awkwardly onto the available editing functions (e.g., editors are line-oriented string processors whereas users conceptualize the editing material in *word* or *phrase* units, often as a single continuous stream, rather than as a sequence of arbitrary lines); (2) the display of searched-for, modified, or formatted editing material is not consonant with the requirements of the user's editing task (e.g., multiple occurrences of a sought-for item are not displayed simultaneously but sequentially, modified material is not displayable as distinct from new or prior unmodified material, and the user typically is forced to achieve formatting of the material by control instructions within one display mode and then invoke another environment to portray the formatted results).

7.2.4.2.2 *File Manipulation:* Users should be able to perform the following file operations in a natural way: (1) naming, (2) describing, (3) storing, (4) copying, (5) distributing, (6) annotating, and (7) retrieving. Compared to the normal people-operated facilities and practices for file manipulation in industry and Government, computer systems support these actions at best awkwardly and at worst not at all:

(1). *Naming* is usually severely restricted by limiting the name to one or two words, with each word limited in its number of characters. Further, no provisions are made for *natural* language naming practices, such as the use of prenominal qualification in which a variable number of descriptors are inserted to the *left* of the head noun -- e.g., "Douglas vs. Perry Decision", "Irkutsk Reconnaissance Report", "Contractor's Final Report", etc.

(2). Facilities for *Describing* file content, topic, source, purpose, etc., are almost never made available to permit the user to establish any kind of reference catalog. Given the severe naming restrictions, and the filing restrictions discussed below, the user may have to resort to some laborious process of examining the contents of a series of files to locate the desired file.

(3). Support for *Storing* files is limited to a one-level structure, in contrast to natural hierarchical storage of files by main category, sub-category, etc. Only by clever and contorted file-naming can a user achieve a modicum of aggregate clustering of similar files. In contrast, a reasonable storing system would not only provide for categorized storing, but would also permit the insertion of the file-name in other places in the hierarchy when the file is pertinent to more than one subject. Further, re-sorting of files into existing categories and re-definition of category structures would also be supported.

(4). *Copying* a file in its entirety usually suffers only from awkward command syntax; however, the copying of limited portions or the extraction of exceprts requires the user to invoke a lengthy series of editing commands.

(5). *Distributing* files again usually suffers only from unnatural syntax *as long as* the user is privy to the computer system's naming scheme for the intended recipients and their whereabouts. However, it is frequently the case that John Smith's *computer* name is something entirely different, and his location -- as a "node" on a computer network -- could be anything; and computer systems and security controls often make it very difficult to determine these. Further, the system is ruthlessly intolerant of normal person/place references, allowing no synonyms, no abbreviations, no misspellings, and not even some form of polite assistance dialog.

(6). The highly common practice of *Annotating* files or documents is completely unsupported by computer systems, where comments can be added to a document without actually modifying it in an editing sense. While developing the software to accomplish this might not be trivial, it is straight-forward conceptually, involving some facility for overlaying comments (e.g., between the document's double-spaced lines) and making it possible to separate the comments from the document (e.g., using differential display brightnesses for the comments vs. the document and providing for the simultaneous overlay display of two different files).

(7). Finally, computer systems typically provide no other mechanism for *Retrieving* files other than by file-name. While the user may have access to an information retrieval system which can search a document data-base for, at most, key word-strings, these systems do not permit the user to search his own files, either as an included or an exclusive set. Most serious computer users -- particularly programmers -- will have personal files numbering in the hundreds and may further be potentially interested in many times this number of other people's files, or system files. Given the naming, describing, and filing restrictions discussed above, retrieving pertinent files can be a major undertaking. (We suspect that there is a great deal of re-writing of programs or other kinds of files just because it can be so difficult to find the original ones).

7.2.4.2.3 *Information-Partitioned Displays:* Two technological trends lead us to emphasize this third area as important: (1) the shift from typewriter and teletype terminals to hign information-density display screens, and (2) the increasing availability of and requirement for distributed remote-terminal networks.

The development of large-screen displays, in particular, provides a number of opportunities for increasing the productivity of computer users. Probably the least important advantage of large-screens is the capability to display a greater amount of information of the same kind -- e.g., more lines of a program or text. Much more interesting is the possibility of displaying *different* kinds of information simultaneously. For example, program debugging and testing could be enhanced by facilities which incrementally executed the program, line by line, showing the program segment surrounding the (highlighted) executed-line in one portion of the screen while displaying the results of the execution in another portion. Similarly, two versions of a program can be simultaneously displayed for comparison. In general, large-screen displays make it possible to show *many* kinds of information at the same time. Since it is the nature of many tasks that several different information sources may be simultaneously relevant, information-partitioned displays can better and more directly support these tasks. In addition, humans are also notorious multi-task processors, engaging in several activities of different kinds all at the same time -- e.g., monitoring a program's execution as the primary task while simultaneously attending to written material, answering questions, checking the time or appointment calendar, making notes, etc. Many of these secondary activities could be supported directly on-line by partitioning the screen into separate fields -- each assigned for a different task -- and giving the user separate and easy control over each field.

The second factor, that of distributed terminals, implies that users may be physically *remote* from each other, communicating via the computer network. Simple messages and transmission of other information between users is currently supported and presents no behavioral problems (other than those discussed above in section 7.2.4.2.2). However, it is not possible at present for remote users to work simultaneously on the *same* task with any degree of effectiveness. Consider the situation in which two users wish to "discuss" the same material -- e.g., a program. Were they at the same physical location that natural mode of discussion would be to place the program in view of both, with each taking turns pointing to the material and making various annotations (which might remain or else be erased). Concurrent with the references to the commonly-displayed material would be a running

dialogue of discussion. Such a "discussion" is inconceivable between *remote* users at the present time, since no communication facilities exist (outside of small exploratory research efforts) to support this kind of interaction. Nevertheless, large-screen displays could be appropriately partitioned, and suitable software developed, to permit such remote joint-work interaction in a manner very similar to natural situations.

7.3 *RECOMMENDATIONS:*

7.3.1 *Routine and Problem-solving Tasks* -- The sections on these two classes of tasks generally proposed that a clear understanding of the application to be programmed could provide the basis for much more effective design, program development, programming, and ultimate usability.

We feel that this proposal has sufficient merit to warrant its evaluation as a general methodology for improving the productivity and quality of application programming tasks. We specifically recommend that the method be tested, on a controlled experimental basis, by choosing a type of routine-task application of particular concern to the Navy and monitoring program development with and without this methodology. The four criteria can be employed to identify routine-tasks, and the model can provide the basis for further determining the specific human activities involved in the task, which activities are to be supported by the programmed application.

If the method appears feasible, we further recommend that research be performed to elaborate the routine-task model and also to begin further definition of a problem-solving typology for eventual use in the same manner.

7.3.2 *Editing* -- We believe that the development of high-level task-oriented editing facilities would be of enormous value in increasing the effectiveness of programming, and the benefits could be realized immediately. If we were asked to make only a *single* recommendation for better programming tools, it would be to provide programming and text editors which correct the deficiencies we have noted above, in our review (12), and in our recommendation for supporting software desgin (see Section 5.3.2.3).

We recommend that an independent and comprehensive review of the usage problems with editors be conducted to verify and augment our observations. Subsequently, the actual development of optimized editors should be supported.

7.3.3 *File-humanipulation and Information-partitioned displays* -- We recommend that the file-humanipulation problems pointed out above be further assessed and development efforts begun to develop better facilities.

As for information-partitioned displays, we believe that basic research is necessary to determine the limiting parameters of this concept as a function of the user and task -- e.g., the number of independent fields, the method of differentiating important from less-important material -- as well as to determine the operating protocols governing remote user communication we lead to effective cooperative work. We note that this concept seems ideally suited to the Navy's requirements for tactical decision-support and communication interfaces, and we recommend that experimental work be conducted, in that context, to evalute the potential of this method of information-display.

8. BIBLIOGRAPHY

NO.	YEAR	IBM NO.	AUTHOR(S)	TITLE
1.	1972	RC4169	Boies	User behavior on an interactive computer system. (IBM Systems Journal, 1974, 13, 2-18)
2.	1973	RC4280	Miller	Programming by non-programmers. (International Journal of Man-Machine Studies, 1974,6,237-260)
3.	1973	RC4472	Boies/Spiegel	A behavioral analysis of programming: On the use of interactive debugging facilities.
4.	1974	RC4956	Durding/Becker/ Gould	Data organization. (Human Factors, 1977, 19, 1-14)
5.	1974	RC5137	Miller/Becker	Programming in natural english. (Human Factors, forthcoming)
6.	1975	RC5279	Gould/Ascher	Use of an IQF-like query language by non-programmers.
7.	1975	------	Miller	Naive programmer problems with specification of transfer-of-control.(National Computer Conf., Anaheim, 1975, 44, 657-663)
8.	1976	RC5866	Thomas	Quantifiers and question-asking.
9.	1976	RC5882	Thomas	A method for studying natural language dialogue.
10.	1976	RC5943	Gould/Lewis/ Becker	Writing and following procedural, descriptive, and restricted syntax, language instructions.
11.	1976	RC6199	Chodorow/Miller	The interpretation of temporal order in coordinate conjunction.
12.	1976	RC6326	Miller/Thomas	Behavioral Issues in the use of interactive systems (Internat'l Journal of Man-Machine Studies, 1977, 9, 509-536)
13.	1976	------	Miller	Natural Language Procedures: Guides for programming language. International Ergonomics Assoc., Univ. of Maryland, July, 1976.

8. BIBLIOGRAPHY (CONTINUED)

NO.	YEAR	IBM NO.	AUTHOR(S)	TITLE
14.	1977	RC6468	Thomas/Lyon/ Miller	Aids for problem solving.
15.	1977	RC6581	Thomas	A design-interpretation analysis of natural english with applications to human-computer interaction.
16.	1977	RC6627	Lyon/Thomas	Predicting insufficient learning of a complex procedure.
17.	1977	RC6702	Thomas/Malhotra/ Carroll	An experimental investigation of the design process.
18.	1978	RC6975	Carroll/Thomas/ Malhotra	Presentation and representation in design problem solving.
19.	1978	RC7072	Evans/Miller	STARCAT: A system to analyze interactive CMS performance.
20.	1978	RC7078	Carroll/Thomas/ Miller	Aspects of solution structure in design problem solving.
21.	1978	RC7081	Carroll/Thomas/ Malhotra	A clinical-experimental analysis of design problem solving.
22.	1978	RC7082	Malhotra/Thomas/ Carroll/Miller	Cognitive processes in design.

9. *PERSONNEL ASSOCIATED WITH THE CONTRACT RESEARCH.**

IBM Principal and Co-principal Investigators

Dr. Lance Miller (Principal)
Dr. Stephen Boies (Co-Principal)
Dr. John Gould (Co-Principal)
Dr. John Thomas (Co-Principal)

Other IBM Personnel

Dr. Clayton Lewis
Dr. Ashok Malhotra
Dr. John Parkman
Mr. Robert Ascher

1-2 year Visiting Post-Doctoral (PD) and Graduate Student (GS)

Dr. John Carroll (PD)
Dr. Martin Chodorow (PD)
Dr. Bruce Durding (PD)
Dr. Donald Lyon (PD)
Dr. K. W. Scholz (PD)
Mr. Curtis Becker (GS)
Mr. Brian Madden (GS)
Mr. Murray Spiegal (GS)

Consultants
Dr. K. W. Scholz
Mr. Edward Schulman

* Only the Principal was associated continuously with the contract research. Other personnel were associated for shorter periods and at different times. The status of the personnel is given as of the time they were associated with the contract.

Office of Naval Research, Code 455
Technical Reports Distribution List

Director, Engineering Psychology
Programs, Code 455
Office of Naval Research
800 North Quincy Street
Arlington, VA 22217 (5 cys)

Defense Documentation Center
Cameron Station
Alexandria, VA 22314 (12 cys)

Dr. Stephen Andriole
Director, Cybernetics Technology
Office
Advanced Research Projects Agency
1400 Wilson Blvd
Arlington, VA 22209

Cmdr. Paul R. Chatelier
OUSDRE (E&LS) ODDR&E
Pentagon, Room 3D129
Washington, D.C. 20301

Mr. Kin B. Thompson
Technical Director
Information Systems Division
OP-81T
Office of the Chief of Naval
Operations
Washington, D.C. 20350

Director, Information Systems
Program, Code 437
Office of Naval Research
800 North Quincy Street
Arlington, VA 22217

Director, Physiology Program
Code 441
Office of Naval Research
800 North Quincy Street
Arlington, VA 22217

DARPA
Director, IPTO
1400 Wilson Blvd.
Arlington, VA 22209

Commanding Officer
ONR Branch Office
ATTN: Dr. J. Lester
Bldg 114, Section D
666 Summer Street
Boston, MA 02210

Commanding Officer
ONR Branch Office
ATTN: Dr. Charles Davis
536 South Clark Street
Chicago, IL 60605

Commanding Officer
ONR Branch Office
ATTN: Dr. E. Gloye
1030 East Green Street
Pasadena, CA 91106

Dr. Bruce McDonald
Office of Naval Research
Scientific Liaison Group
American Embassy, Room A-407
APO San Francisco 96503

Director, Naval Research Laboratory
Technical Information Division
Code 2627
Washington, D.C. 20375 (6 cys)

Dr. Bruce Wald
Naval Research Laboratory
Communication Sciences Division
Attn: Code 7500
Washington, D.C. 20375

Dr. Robert G. Smith
Office of the Chief of Naval
Operations, OP987H
Personnel Logistics Plans
Department of the Navy
Washington, D.C. 20350

Mr. Arnold Rubinstein
Naval Material Command
NAVMAT 08T24
Department of the Navy
Washington, D.C. 20360

7.2.2.1 *Characteristics of routine tasks* -- We define tasks as being "routine" when they possess all of the following four characteristics. Routine tasks --

Commander
Naval Air Systems Command
Human Factors Programs,AIR 340F
Washington, D.C. 20361

Commander
Naval Air Systems Command
Crew Station Design,AIR 5313
Washington, D.C. 20361

Commander
Naval Electronics Systems
Command
Human Factors Engineering Branch
Code 4701
Washington, D.C. 20360

Dr. James Curtin
Naval Sea Systems Command
Personnel & Training Analyses Office
NAVSEA 074C1
Washington, D.C. 20362

Dr. Arthur Bachrach
Behavioral Sciences Department
Naval Medical Research Institute
Bethesda, MD 20014

Dr. George Moeller
Human Factors Engineering Branch
Submarine Medical Research
Laboratory
Naval Submarine Base
Groton, CT 06340

Chief, Aerospace Psychology Division
Naval Aerospace Medical Institute
Pensacola, FL 32512

Mr. Phillip Andrews
Naval Sea Systems Command
NAVSEA 0341
Washington, D.C. 20362

Bureau of Naval Personnel
Special Assistant for Research
Liaison
PERS-OR
Washington, D.C. 20370

Navy Personnel Research and
Development Center
Management Support Department
Code 210
San Diego, CA 92152

Dr. Fred Muckler
Navy Personnel Research and
Development Center
Manned Systems Design, Code 311
San Diego, CA 92152

Mr. Mel Moy
Navy Personnel Research and
Development Center
Code 305
San Diego, CA 92152

Mr. A. V. Anderson
Navy Personnel Research and
Development Center
Code 302
San Diego, CA 92152

CMDR P. M. Curran
Human Factors Engineering Division
Naval Air Development Center
Warminster, PA 18974

LCDR William Moroney
Human Factors Engineering Branch
Code 1226
Pacific Missile Test Center
Point Mugu, CA 93042

Human Factors Section
Systems Engineering Test
Directorate
U.S. Naval Air Test Center
Patuxent River, MD 20670

Dr. John Silva
Man-System Interaction Division
Code 823, Naval Ocean Systems Center
San Diego, CA 92152

Human Factors Engineering Branch
Naval Ship Research and Development
Center, Annapolis Division
Annapolis, MD 21402

Naval Training Equipment Center
ATTN: Technical Library
Orlando, FL 32813

Human Factors Department
Code N215
Naval Training Equipment Center

reference to a higher authority. The output of the decision is either termination of the procedure or identification of an appropriate pattern-procedure match. To the extent that the decision/information-gathering procedures invoked are themselves routine tasks,

Orlando, FL 32813

Dr. Alfred F. Smode
Training Analysis and Evaluation Group
Naval Training Equipment Center
Code N-OOT
Orlando, FL 32813

Dr. Gary Poock
Operations Research Department
Naval Postgraduate School
Monterey, CA 93940

Dr. A. L. Slafkosky
Scientific Advisor
Commandant of the Marine Corps
Code RD-1
Washington, D.C. 20380

Mr. J. Barber
Headquarters, Department of the
Army, DAPE-PBR
Washington, D.C. 20546

Dr. Joseph Zeidner
Technical Director
U.S. Army Research Institute
5001 Eisenhower Avenue
Alexandria, VA 22333

Dr. Edgar M. Johnson
Organization and Systems
Research Laboratory
U.S. Army Research Institute
5001 Eisenhower Avenue
Alexandria, VA 22333

Technical Director
U.S. Army Human Engineering Labs
Aberdeen Proving Ground
Aberdeen, MD 21005

U.S. Air Force Office of Scientific
Research
Life Sciences Directorate, NL
Bolling Air Force Base
Washington, D.C. 20332

Dr. Donald A. Topmiller
Chief, Systems Engineering Branch
Human Engineering Division
USAF AMRL/HES
Wright-Patterson AFB, OH 45433

Lt. Col. Joseph A. Birt
Human Engineering Division
Aerospace Medical Research Laborator
Wright-Patterson AFB, OH 45433

Air University Library
Maxwell Air Force Base, AL 36112

Dr. Arthur I. Siegel
Applied Psychological Services, Inc.
404 East Lancaster Street
Wayne, PA 19087

Dr. Gershon Weltman
Perceptronics, Inc.
6271 Variel Avenue
Woodland Hills, CA 91364

Dr. Jesse Orlansky
Institute for Defense Analyses
400 Army-Navy Drive
Arlington, VA 22202

Dr. Stanley Deutsch
Office of Life Sciences
HQS, NASA
600 Independence Avenue
Washington, D.C. 20546

Director, National Security Agency
ATTN: Dr. Douglas Cope
Code R51
Ft. George G. Meade, MD 20755

Journal Supplement Abstract Service
American Psychological Association
1200 17th Street, N.W.
Washington, D.C. 20036 (3 cys)

Dr. William A. McClelland
Human Resources Research Office
300 N. Washington Stret
Alexandria, VA 22314

Dr. Bill Curtis
General Electric Company
Information Systems Programs
1755 Jefferson Davis Highway
Arlington, VA 22202

Director, Human Factors Wing
Defence & Civil Institute of
Environmental Medicine
Post Office Box 2000

4. *Place-keeping* -- For tasks involving the possibility of recursively invoking subprocedures assistance could be provided by automatically keeping track of the hierarchical status of suspended, present, and remaining tasks, and making this record

Downsville, Toronto, Ontario
CANADA

Dr. A. D. Baddeley
Director, Applied Psychology Unit

Medical Research Council
15 Chaucer Road
Cambridge, CB2 2EF
ENGLAND